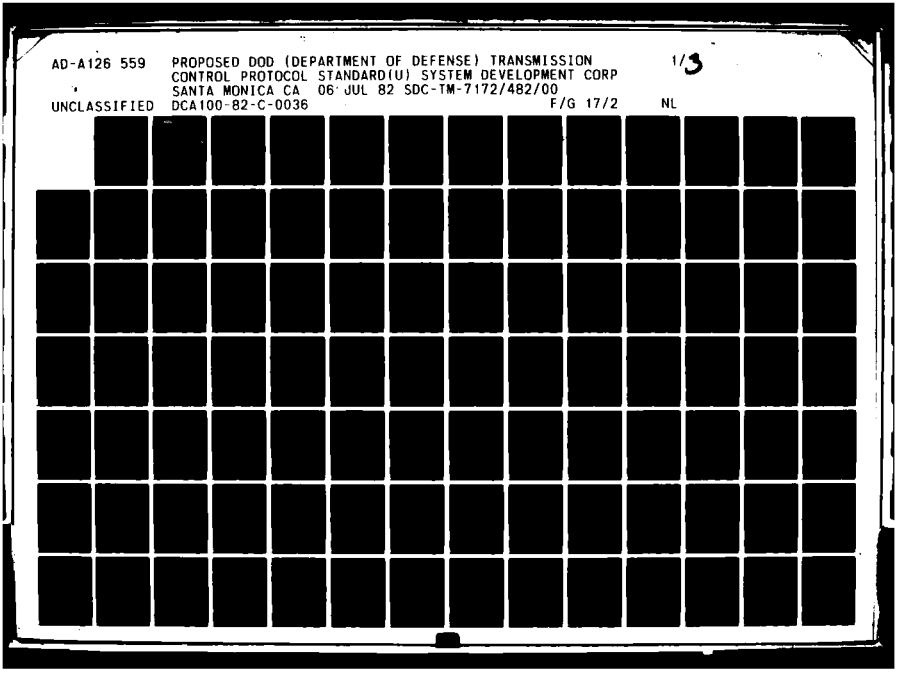
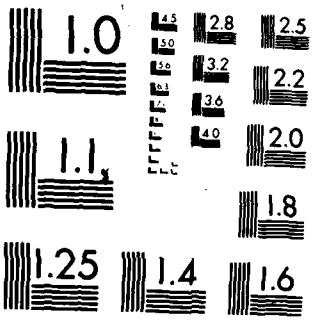

AD-A126 559 PROPOSED DOD (DEPARTMENT OF DEFENSE) TRANSMISSION CONTROL PROTOCOL STANDARD(U) SYSTEM DEVELOPMENT CORP SANTA MONICA CA 06 JUL 82 SDC-TM-7172/482/00
1/3
UNCLASSIFIED DCA100-82-C-0036 F/G 17/2 NL



SDC
System Development Corporation
2500 Colorado Avenue, Santa Monica, CA 90406, Telephone (213) 820-4111

series base no./vol /reissue
7172/482/00

author
Technical Staff
technical
Carl M. Switzky
release
Gerald D. Cole
for
Charles A. Savant

date
7/6/82

TM a working paper

This document was produced by
System Development Corporation in performance of Contract DCA100-82-C-0036

DCEC PROTOCOL STANDARDIZATION PROGRAM

PROPOSED DoD TRANSMISSION CONTROL PROTOCOL STANDARD

July 1982

ABSTRACT

This document specifies the Transmission Control Protocol (TCP), a reliable connection-oriented transport protocol for use in packet-switched communication networks and internetworks. The document includes an overview with a model of operation, a description of services offered to users, and a description of the architectural and environmental requirements. The protocol service interfaces and mechanisms are specified using an extended state machine model.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 7172/482/00	2. GOVT ACCESSION NO. AD-A126559	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Proposal DoD Transmission Control Protocol Standard		5. TYPE OF REPORT & PERIOD COVERED interim technical report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) SDC Technical Staff		8. CONTRACT OR GRANT NUMBER(s) DCA100-82-C-0036
9. PERFORMING ORGANIZATION NAME AND ADDRESS System Development Corporation 2500 Colorado Ave. Santa Monica, CA 90406		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS P.E. 33126K Task 1052.558
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Communications Engineering Center Switched Networks Engineering Directorate 1860 Wiehle Ave., Reston, VA 22090		12. REPORT DATE Jul 82 13. NUMBER OF PAGES 190
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* N/A		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for Public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

N/A

18. SUPPLEMENTARY NOTES

This document represents results of interim studies which are continuing at the DCEC of DCA.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Protocols, Data Communications, Data Networks, Protocol Standardization, Transmission Control Protocol

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

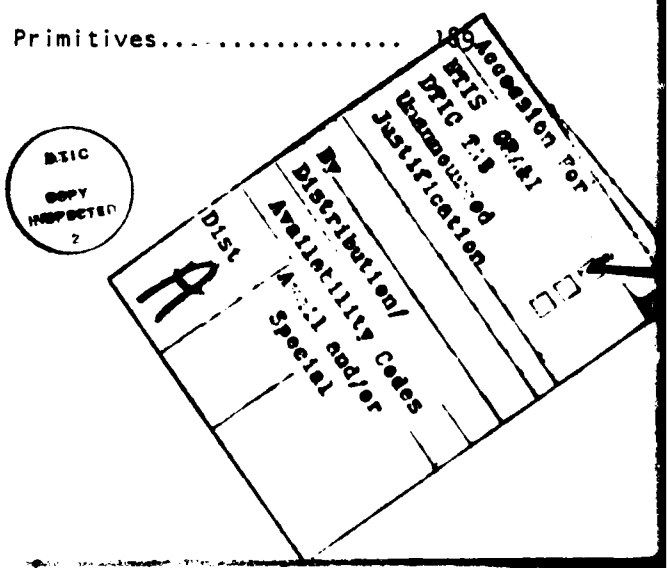
This document specifies the Transmission Control Protocol (TCP), a reliable connection-oriented trans port protocol for use in packet-switched communication networks and internetworks. The document includes an overview with a model of operation, a description of services offered to users, and a description of the architectural and environmental requirements. The protocol service interfaces and mechanisms are specified using an extended state machine model.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

CONTENTS



1. OVERVIEW

This document specifies the Transmission Control Protocol (TCP), a reliable connection-oriented transport protocol for use in packet-switched and other communication networks and interconnected sets of such networks. The document is organized into seven sections plus a bibliography and a glossary of terms. This, the first section, establishes TCP's role in the evolving DoD protocol architecture and introduces TCP's major services and mechanisms. The second and third sections more formally specify the services TCP offers to upper layer protocols and the interface through which those services are accessed. Similarly, the next two sections specify the services required of the lower layer protocol and the lower interface. The sixth section specifies the mechanisms supporting the TCP services. The seventh section outlines the functionality required of the execution environment for successful TCP operation. The reader is assumed to be familiar with the proposed DoD protocol architecture which defines a layered model of network communications systems[6].

This document incorporates the organization and specification techniques presented in the Protocol Specification Guidelines[7]. One goal of the guidelines is to avoid assuming a particular system configuration. As a practical matter, the distribution of protocol layers to specific hardware configurations will vary. For example, many computer systems are connected to networks via front-end computers which house TCP and lower layer protocol software. Although appearing to focus on TCP implementations which are co-resident with the upper and lower layer protocols, this specification can apply to any configuration given appropriate inter-layer protocols to bridge hardware boundaries.

TCP is designed to provide reliable communication between pairs of processes in logically distinct hosts on networks and sets of interconnected networks. Thus, TCP serves as the basis for DoD-wide inter-process communication in communication systems. TCP will operate successfully in an environment where the loss, damage, duplication, or misorder data, and network congestion can occur. This robustness in spite of unreliable communications media makes TCP well suited to support military, governmental, and commercial applications.

TCP appears in the DoD protocol hierarchy at the transport layer. Here, TCP provides connection-oriented data transfer that is reliable, ordered, full-duplex, and flow controlled. TCP is designed to support a wide range of upper layer protocols (ULPs). The ULPs can channel continuous streams of data through TCP for delivery to peer ULPs. TCP breaks the streams into portions which are encapsulated together with appropriate addressing and control information to form a segment--the unit of exchange between peer TCPs. In turn, TCP passes segments to the network layer for transmission through the communication system to the peer TCP.

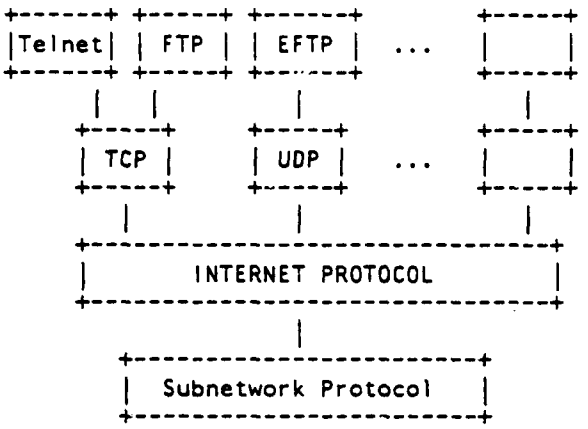


Figure 1. Example Host Protocol Hierarchy

The network layer provides for data transfer between hosts attached to a communication system. Such systems may range from a single network to interconnected sets of networks forming an internetwork. The minimum required data transfer service is limited; data may be lost, duplicated, misordered, or damaged in transit. As part the transfer service though, the network layer must provide global addressing, handle routing, and hide network-specific characteristics. As a result, upper layer protocols (including TCP) using the network layer may operate above a wide spectrum of subnetwork systems ranging from hard-wire connections to packet-switched or circuit-switched subnets. Additional services the network layer may provide include selectable levels of transmission quality such as precedence, reliability, delay, and throughput. The network layer also allows data labelling, needed in secure environments, to associate security information with data.

TCP was specifically designed to operate above the Internet Protocol (IP) which supports the interconnection of networks[8]. IP's internet datagram service provides the functionality described above. Originally, TCP and IP were developed as a single protocol providing resource sharing across different packet networks[1]. The need for other transport protocols to use IP's services led to their specification as two distinct protocols[3,4].

TCP builds its services on top of the network layer's potentially unreliable ones with mechanisms such as error detection, positive acknowledgments, sequence numbers, and flow control. These mechanisms require certain addressing and control information to be initialized and maintained during data transfer. This collection of information is called a TCP connection. The

following paragraphs describe the purpose and operation of the major TCP mechanisms.

TCP uses a positive acknowledgement with retransmission (PAR) mechanism to recover from the loss of a segment by the lower layers. The strategy with PAR is for a sending TCP to retransmit a segment at timed intervals until a positive acknowledgement is returned. The choice of retransmission interval affects efficiency. An interval that is too long reduces data throughput while one that is too short floods the transmission media with superfluous segments. A discussion of retransmission intervals is presented in [2]. In TCP, the timeout is expected to be dynamically adjusted to approximate the segment round-trip time plus a factor for internal processing, otherwise performance degradation may occur [9].

TCP uses a simple checksum to detect segments damaged in transit. Such segments are discarded without being acknowledged. Hence, damaged segments are treated identically to lost segments and are compensated for by the PAR mechanism.

TCP assigns sequence numbers to identify each octet (an eight bit byte) of the data stream. These enable a receiving TCP to detect duplicate and out-of-order segments. Sequence numbers are also used to extend the PAR mechanism by allowing a single acknowledgment to cover many segments worth of data. Thus, a sending TCP can still send new data although previous data has not been acknowledged.

TCP's flow control mechanism enables a receiving TCP to govern the amount of data dispatched by a sending TCP. The mechanism is based on a "window" which defines a contiguous interval of acceptable sequence numbered data. As data is accepted, TCP slides the window upward in the sequence number space. This window is carried in every segment enabling peer TCPs to maintain up-to-date window information.

TCP employs a multiplexing mechanism to allow multiple ULPs within a single host and multiple proccesses in a ULP to use TCP simultaneously. This mechanism associates identifiers, called ports, to ULP's processes accessing TCP services. A ULP connection is uniquely identified with a socket, the concatenation of a port and an internet address. Each connection, is uniquely named with a socket pair. This naming scheme allows a single ULP to support connections to multiple remote ULPs. ULPs which provide popular resources are assigned permanent sockets, called well-known sockets. Some well-known sockets currently used in DoD networks are published in in [5].

When two ULPs wish to to communicate, they instruct their TCP's to initialize and synchronize the mechanism information on each to open the connection. However, the potentially unreliable network layer can complicate the process of synchronization. Delayed or duplicate segments from previous connection attempts might be mistaken for new ones. A handshake procedure with clock based sequence numbers is used in connection opening to reduce the possibility of such false connections. In the simplest handshake, the TCP pair synchronizes sequence numbers by exchanging three segments, thus the name three-way handshake. The scenario following the overview depicts this exchange. The

procedure will be discussed more fully in the mechanism descriptions, Section 6.1. A comprehensive study of connection establishment and related connection management issues can be found in [10].

A ULP can open a connection in one of two modes, passive or active. With a passive open a ULP instructs its TCP to be "receptive" to connections with other ULPs. With an active open a ULP instructs its TCP to actively initiate a three-way handshake to connect to another ULP. Usually, an active open is targeted to a passive open. This active/passive model supports server-oriented applications where a permanent resource, such as a data-base management process, can always be accessed by remote users. However, the three-way handshake also coordinates two simultaneous active opens to open a connection.

Over an open connection, the ULP-pair can exchange a continuous stream of data in both directions. Normally, TCP transparently groups the data into TCP segments for transmission at its own convenience. However, a ULP can exercise a "push" service to force TCP to package and send data passed up to that point without waiting for additional data. This mechanism is intended to prevent possible deadlock situations where a ULP waits for data internally buffered by TCP. For example, an interactive editor might wait forever for a single input line from a terminal. A push will force data through the TCPs to the awaiting process. TCP also provides a means for a sending ULP to indicate to a receiving ULP that "urgent" data appears in the upcoming data stream. This urgent mechanism can support, for example, interrupts or breaks.

When data exchange is complete the connection can be closed by either ULP to free TCP resources for other connections. Connection closing can happen in two ways. The first, called a graceful close, is based on the three-way handshake procedure to complete data exchange and coordinate closure between the TCPs. The second, called an abort, does not allow coordination and may result in loss of unacknowledged data.

1.1 SCENARIO

The following scenario provides a walk-through of a connection opening, data exchange, and a connection closing as might occur between the data base management process and user mentioned above. The scenario glosses over many details to focus on the three-way handshake mechanism in connection opening and closing, and the positive acknowledgement with retransmission mechanism supporting reliable data transfer. Although not pictured, the network layer transfers the information between the TCPs. For the purposes of this scenario, the network layer is assumed not to damage, lose, duplicate, or change the order of data unless explicitly noted.

The scenario is organized into three parts:

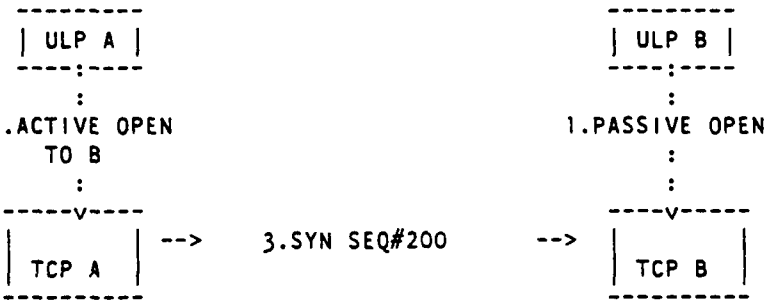
a. A simple connection opening in steps 1-7.

b. Two-way data transfer in steps 8-17.

c. A graceful connection close in steps 18-25.

The following notation is used in the diagrams:

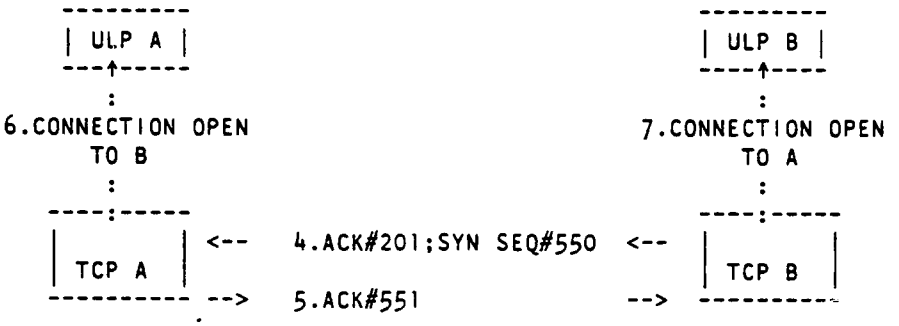
```
<--  SEQ# 200  <--        depicts information exchange
-->  ACK# 201  -->        between peer TCPs

    :          ↑          depicts information passing
SEND DATA      :          across the interface between
    :      DELIVER DATA   a ULP and its TCP
    v          :
```



1. ULP B (the DB manager) issues a PASSIVE OPEN to TCP B to prepare for connection attempts from other ULPs in the system.

2. ULP A (the user) issues an ACTIVE OPEN to open a connection to ULP B.

3. TCP A sends a segment to TCP B with an OPEN control flag, called a SYN, carrying the first sequence number (shown as SEQ#200) it will use for data sent to B.

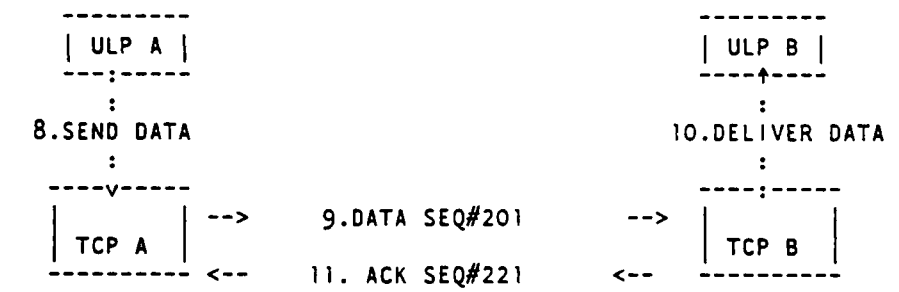


4. TCP B responds to the SYN by sending a positive acknowledgement ACK, marked with next sequence number expected from TCP A. In the ne segment, TCP B sends its own SYN with the first sequence number r its data (SEQ#550).

5. TCP A responds to TCP B's SYN with an ACK showing the next sequence number expected from B.

6. TCP A now informs ULP A that a connection is open to ULP B.

7. Upon receiving the ACK, TCP B informs ULP B that a connection has been opened to ULP A.

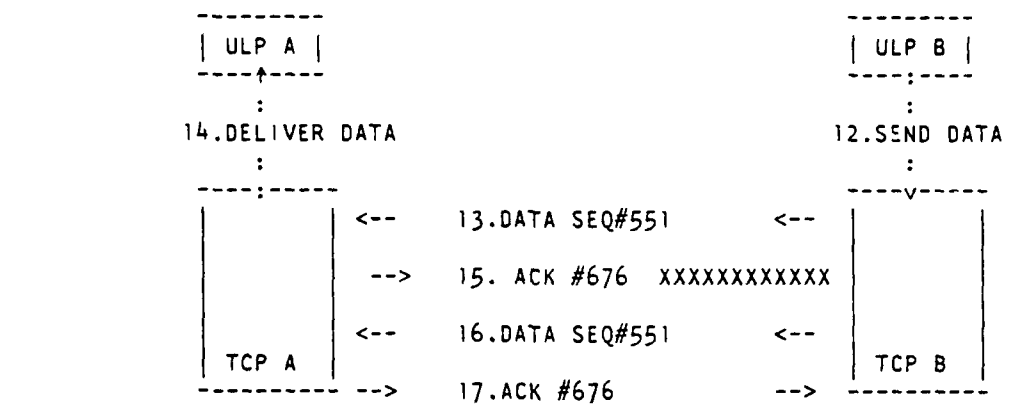


8. ULP A passes 20 octets of data to TCP A for transfer across the open connection to ULP B.

9. TCP A packages the data in a segment marked with current "A" sequence number.

10. After validating the sequence number, TCP B accepts the data and delivers it to ULP B.

11. TCP B acknowledges all 20 octets of data with the ACK set to the sequence number of the next data octet expected.



12. ULP B passes 125 bytes of data to TCP B for transfer to ULP A.

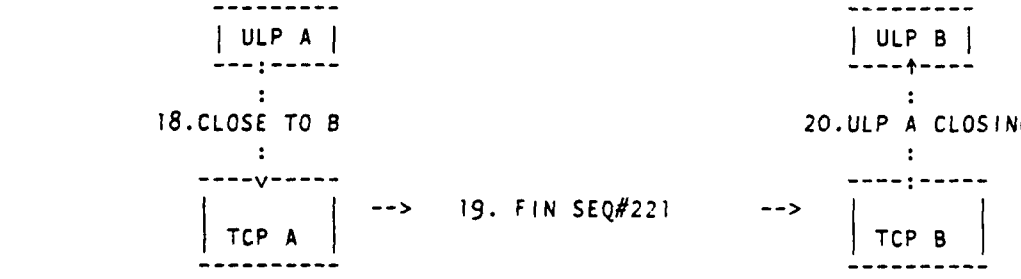
13. TCP B packages the data in a segment marked with the "B" sequence number.

14. TCP A accepts the segment and delivers the data to ULP A.

15. TCP A returns an ACK of the received data marked with the sequence number of the next expected data octet. However, the segment is lost by the network and never arrives at TCP B.

16. TCP B times out waiting for the lost ACK and retransmits the segment. TCP A receives the retransmitted segment, but discards it because the data from the original segment has already been accepted. However, TCP A re-sends the ACK.

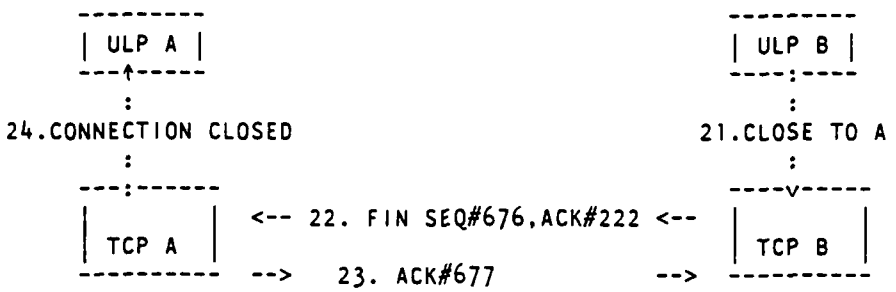
17. TCP B gets the second ACK.



18. ULP A closes its half of the connection by issuing a CLOSE to TCP A.

19. TCP A sends a segment marked with a CLOSE control flag, called a FIN, to inform TCP B that ULP A will send no more data.

20. TCP B gets the FIN and informs ULP B that ULP A is closing.



21. ULP B completes its data transfer and closes its half of the connection.

22. TCP B sends an ACK of the first FIN and its own FIN to TCP A to show ULP B's closing.

23. TCP A gets the FIN and the ACK, then responds with an ACK to TCP B.

24. TCP A informs ULP A that the connection is closed.

25. (Not pictured) TCP B receives the ACK from TCP A and informs ULP B that the connection is closed.

2. SERVICES PROVIDED TO UPPER LAYER

This section describes the services offered by the Transmission Control Protocol to upper layer protocols (ULPs). The goals of this section are to provide the motivation for protocol mechanisms and to provide ULPs with a definition of the functions provided by this protocol.

The services provided by TCP can be organized as follows:

- o multiplexing service
- o connection management services
- o data transport service
- o error reporting service

A description of each service follows.

2.1 MULTIPLEXING SERVICE

TCP shall provide services to multiple pairs of processes within upper layer protocols. A process within a ULP using TCP services shall be identified with a "port". A port, when concatenated with an internet address, forms a socket which uniquely names a ULP throughout the internet. TCP shall use the pair of sockets corresponding to a connection to differentiate between multiple users.

2.2 CONNECTION MANAGEMENT SERVICE

TCP shall provide data transfer capabilities, called connections, between pairs of upper layer protocols. A connection provides a communication channel between two ULPs. Characteristics of data transfer are specified in the data transfer service description. Connection management can be broken into three phases: connection establishment, connection maintenance, and connection termination.

2.2.1 Connection Establishment

TCP shall provide a means to open connections between ULP-pairs. Connections are endowed with certain properties that apply for the lifetime of the connection. These properties, including security and precedence levels, are specified by the ULPs at connection opening. Connections can be opened in one of two modes: active or passive.

TCP shall provide a means for a ULP to actively initiate a connection to another ULP uniquely named with a socket. TCP shall establish a connection to the named ULP if:

1. no connection between the two named sockets already exists,
2. internal TCP resources are sufficient,

3. the other ULP exists, and has

   a. simultaneously executed a matching active open to this ULP,

   b. or, previously executed a matching passive open,

   c. or, previously executed a "global" matching passive open

TCP shall provide a means for a ULP to listen for and respond to active opens from correspondent ULPs. Correspondent ULPs are named in one of two ways:

1. fully specified : A ULP is uniquely named by a socket. A connection is established when a matching active open is executed (as described above) by the named ULP.

2. unspecified : No socket is provided. A connection is established with any ULP executing an matching active open naming this ULP.

2.2.2 Connection Maintenance

TCP shall maintain established connections supporting the data transfer service described in Section 2.3. And, TCP shall provide a means for a ULP to acquire current connection status with regard to connection name, data transfer progress, and connection qualities.

2.2.3 Connection Termination

TCP shall provide a means to terminate established connections and nullify connection attempts. Established connections can be terminated in two ways:

1. Graceful Close : Both ULPs close their side of the duplex connection, either simultaneously or sequentially, when data transfer is complete. TCP shall coordinate connection termination and prevent loss of data in transit as promised by the data transfer service.

2. Abort : One ULP independently forces closure of the connection. TCP shall not coordinate connection termination. Any data in transit may be lost.

2.3 DATA TRANSPORT SERVICE

TCP shall provide data transport over established connections between ULP-pairs. The data transport is full-duplex, timely, ordered, labelled with security and precedence levels, flow controlled, and error-checked. A more detailed description of each of the data transport characteristics follows.

- full-duplex : TCP shall support simultaneous bi-directional data flow between the correspondent ULPs.

- timely: When system conditions prevent timely delivery, as specified by the user timeout, TCP shall notify the local ULP of service failure and subsequently terminate the connection.

- ordered : TCP shall deliver data to a destination ULP in the same sequence as it was provided by the source ULP.

- labelled : TCP shall associate with each connection with the security and precedence levels supplied by the ULPs during connection establishment. When this information is not provided by the ULP-pair, TCP shall assume default levels. TCP shall establish a connection between a ULP-pair only if the security/compartment information exactly matches. If the precedence levels do not match during connection, the higher precedence level is associated with the connection.

- flow controlled : TCP shall regulate the flow of data across the connection to prevent, among other things, internal TCP congestion leading to service degradation and failure.

- error checked : TCP shall deliver data that is free of errors within the probabilities supported by a simple checksum.

TCP shall provide two capabilities to ULPs concerning data transfer over an established connection: data stream push and urgent data signalling.

- data stream push : TCP shall transmit any waiting data up to and including the indicated data portions to the receiving TCP without waiting for additional data. The receiving TCP shall deliver the data to the receiving ULP in the same manner.

- urgent data signalling : TCP shall provide a means for a sending ULP to inform a receiving ULP of the presence of significant, or "urgent," data in the upcoming data stream.

2.4 ERROR REPORTING SERVICE

TCP shall report service failure stemming from catastrophic conditions in the internetwork environment for which TCP cannot compensate.

3. UPPER LAYER SERVICE/INTERFACE SPECIFICATIONS

This section specifies the TCP services provided to upper layer protocols and the interface through which these services are accessed. The first part defines the interaction primitives and interface parameters for the upper interface. The second part contains the extended state machine specification of the upper layer services and interaction discipline.

3.1 INTERACTION PRIMITIVES

An interaction primitive defines the information exchanged between two adjacent protocol layers. Primitives are grouped into two classes based on the direction of information flow. Information passed downward, in this case from a ULP to TCP, is called a service request primitive. Information passed upward, from TCP to the ULP, is called a service response primitive. Interaction primitives need not occur in pairs. That is, a service request does not necessarily elicit a service "response"; a service "response" may occur independently of a service request.

The information associated with an interaction primitive falls into two categories: parameters and data. Parameters describe the data and indicate how it is to be treated. The data itself is neither examined nor modified. The format of the parameters and data is implementation dependent and therefore not specified.

TCP implementations may have different interaction primitives imposed by the execution environment or system design factors. In those cases, the primitives can be modified to include more information or additional primitives can be defined to satisfy system requirements. However, all TCPs must provide at least the information found in the interaction primitives specified below to guarantee that all TCP implementations can support the same protocol hierarchy. Additional primitives that affect the protocol mechanisms may not be used.

3.1.1 Service Request Primitives

The TCP service request primitives enable connection establishment, data transfer, and connection termination. The request primitives are:

1. Unspecified Passive Open,
2. Fully Specified Passive Open,
3. Active Open,
4. Active Open With Data,
5. Send,
6. Allocate,

7. Close,

8. Abort, and

9. Status.

A description and list of parameters for each service request follows. Optional service request parameters are followed by "[optional]."

3.1.1.1 Unspecified Passive Open This service request primitive allows a ULP to listen for and respond to connection attempts from an un-named ULP at a specified security and precedence level. TCP accepts in an Unspecified Passive Open at least the following information:

- source port
- ULP timeout [optional]
- precedence [optional]
- security [optional]

3.1.1.2 Fully Specified Passive Open This service request primitive allows a ULP to listen for and respond to connection attempts from a fully named ULP at a particular security and precedence level. TCP accepts in an Fully Specified Passive Open at least the following information:

- source port
- destination port
- destination address
- ULP timeout [optional]
- precedence [optional]
- security [optional]

3.1.1.3 Active Open This service request primitive allows a ULP to initiate a connection attempt to a named ULP at a particular security and precedence level. TCP accepts in an Active Open at least the following information:

- source port
- destination port
- destination address
- ULP timeout [optional]

- precedence [optional]

- security [optional]

3.1.1.4 Active Open With Data This service request primitive allows a ULP to initiate a connection attempt to a named ULP at a particular security and precedence level accompanied by the specified data. TCP accepts in an Active Open With Data at least the following information:

- source port

- destination port

- destination address

- ULP timeout [optional]

- precedence [optional]

- security [optional]

- data

- data length

- PUSH flag

- URGENT flag

3.1.1.5 Send This service request primitive causes data to be transferred across the named connection. TCP accepts in a Send at least the following information:

- local connection name

- data

- data length

- PUSH flag

- URGENT flag

- ULP timeout [optional]

3.1.1.6 Allocate This service request primitive allows a ULP to issue TCP an incremental allocation for receive data. The parameter, data length, is defined in single octet units. This quantity is the additional number of octets which the receiving ULP is willing to accept. TCP accepts in a Allocate at least the following information:

- local connection name

- data length

3.1.1.7 Close This service request primitive allows a ULP to indicate that it has completed data transfer across the named connection. TCP accepts in a Close at least the following information:

- local connection name

3.1.1.8 Abort This service request primitive allows a ULP to indicate that the named connection is to be immediately terminated. TCP accepts in an Abort at least the following information:

- local connection name

3.1.1.9 Status This service request primitive allows a ULP to query for the current status of the named connection. TCP accepts in an Status at least the following information:

- local connection name

TCP returns the requested status information in a Status Response, defined in Section 3.1.2.7 below.

3.1.2 Service Response Primitives

Several service response primitives are provided to enable TCP to inform the user of connection status, data delivery, connection termination, and error conditions. The response primitives are Open Id, Open Failure, Open Success, Deliver, Closing, Terminate, Status Response, and Error. Each is fully defined in the following paragraphs.

3.1.2.1 Open Id This service response primitive informs a ULP of the local connection name assigned by TCP to the connection requested in one of the previous service requests, Unspecified Open, Fully Specified Open, or an Active Open. TCP provides in an Open Id at least the following information:

- local connection name

- source port

- destination port [if known]

- destination address [if known]

3.1.2.2 Open Failure This service response primitive informs a ULP of the failure of an Active Open service request. TCP provides in an Open Failure at least the following information:

- local connection name

3.1.2.3 Open Success This service response primitive informs a ULP of the completion of one of the Open service requests. TCP provides in an Open Success at least the following information:

- local connection name

3.1.2.4 Deliver This service response primitive informs a ULP of the arrival of data across the named connection. TCP provides in a Deliver at least the following information:

- local connection name
- data
- data length
- URGENT flag

3.1.2.5 Closing This service response primitive informs a ULP that the peer ULP has issued a CLOSE service request. Also, TCP has delivered all data sent by the remote ULP. TCP provides in a Closing at least the following information:

- local connection name

3.1.2.6 Terminate This service response primitive informs a ULP that the named connection has been terminated and no longer exists. TCP generates this response as a result of a remote connection reset, service failure, and connection closing by the local ULP. TCP provides in a Terminate at least the following information:

- local connection name
- description

3.1.2.7 Status Response This service response primitive returns to a ULP the current status information associated with a connection named in a previous Status service request. TCP provides in a Status Response at least the following information:

- local connection name
- source port
- source address
- destination port
- destination address

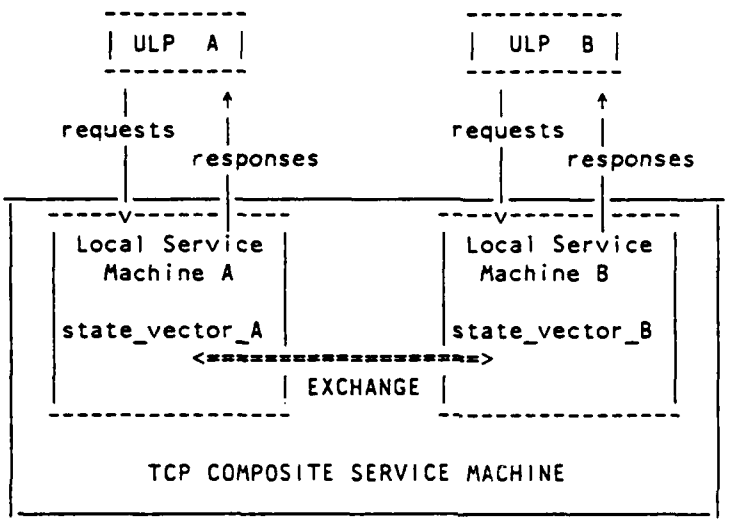
- connection state
- amount of data in octets willing to be accepted by the local TCP
- amount of data in octets allowed to send to the remote TCP
- amount of data in octets awaiting acknowledgement
- amount of data in octets pending receipt by the local ULP
- urgent state
- precedence
- security
- ULP timeout

3.1.2.8 Error This service response primitive informs a ULP of illegal service requests relating to the named connection or of errors relating to the environment. TCP provides in an Error response at least the following information:

- local connection name
- error description

3.2 EXTENDED STATE MACHINE SPECIFICATION SERVICES PROVIDED TO UPPER LAYER

TCP performs in a distributed environment. Hence, an effective model of TCP services can be constructed through the composition of two extended state machines, called local service machines. The following diagram shows a summary of this "split-state" model.



Split State Model of TCP Services

Each local machine is coupled with one ULP of the ULP-pair. Each ULP provides stimuli to its local service machine, in the form of service requests, and receives the resulting reactions, as service responses. Each local machine maintains a complete state record, called a state vector, maintaining a local perspective of the state of the connection. At undetermined intervals, the local machines exchange information (denoted by EXCHANGE) modelling communication delay.

An extended state machine definition is composed of a machine identifier, a state diagram, a state vector, a set of data structures, an event list, and an events and actions correspondence.

3.2.1 Machine Instantiation Identifier

Each local service machine is uniquely identified by the values:

- o port of ULP A
- o address of ULP A
- o port of ULP B

o address of ULP B

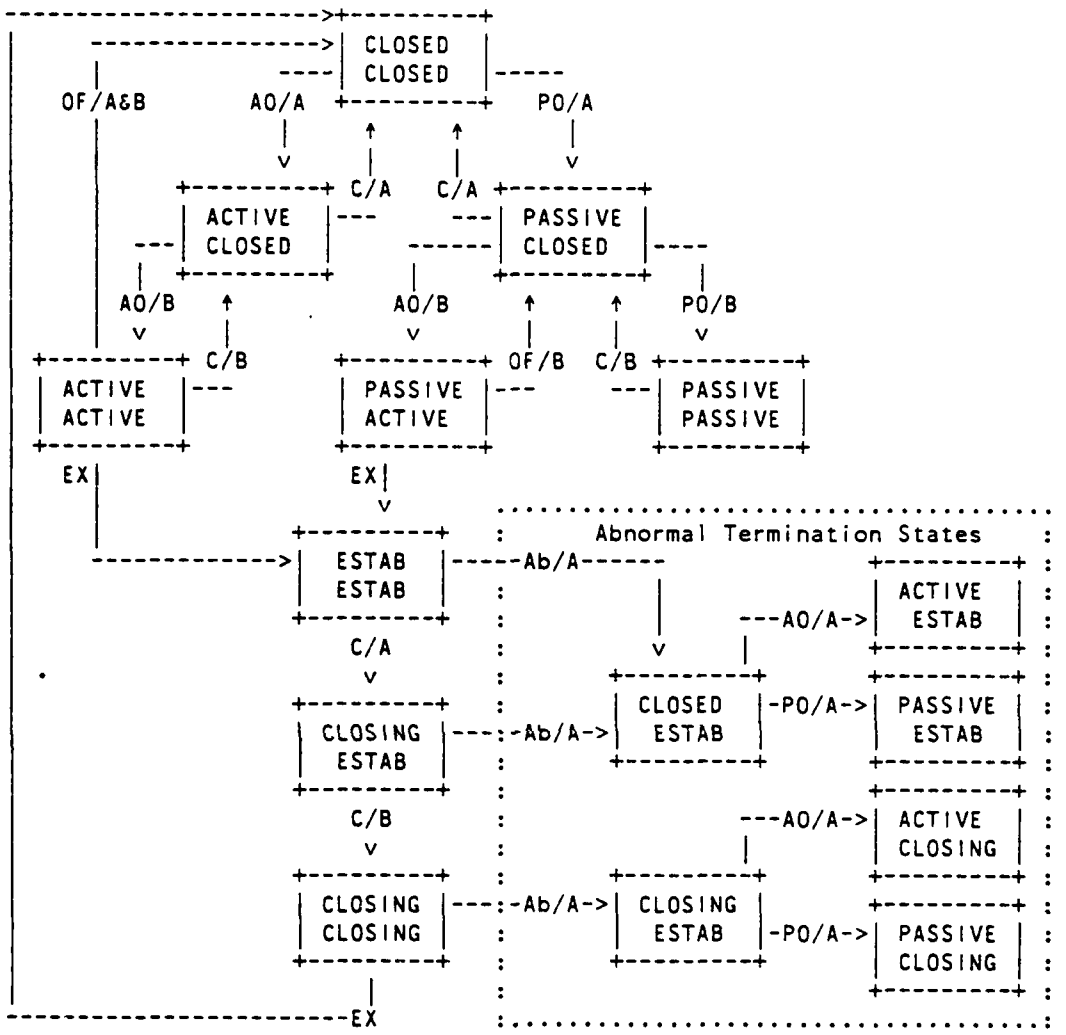
After the first open service request, a TCP uses a shorter name, called a local connection name, to identify a connection in the interactions with its co-resident ULP.

The Unspecified Passive Open service request does not designate the port and address of the remote ULP and such "half-named" service machines are distinguished by local connection name. A fully-named service machine (if it exists) will be connected to a remote open request rather than a half-named service machine with the same source port and source address. Then, if more than one half-named service machine exists, they are connected to matching fully-named remote open requests at random.

3.2.2 State Diagrams

Because of the split-state model presented, both the local service machine state diagram and the composite service machine state diagram are presented.

The following diagram summarizes the service provided by the composite TCP service machine as derived from the composition of two local service machines. The boxes represent the state of the composite service machine; the arrows represent state transitions resulting from the service requests and service responses shown. The "EX" labels represent state changes resulting from the periodic exchanges between local service machines. This diagram serves only as a guide and does not supersede the full definition of the composite service machine in Section 3.2. Abnormal connection termination states are enclosed in the dotted box. These states result from an Abort service request or from TCP service failure.



COMPOSITE TCP SERVICE STATE MACHINE DIAGRAM

LEGEND

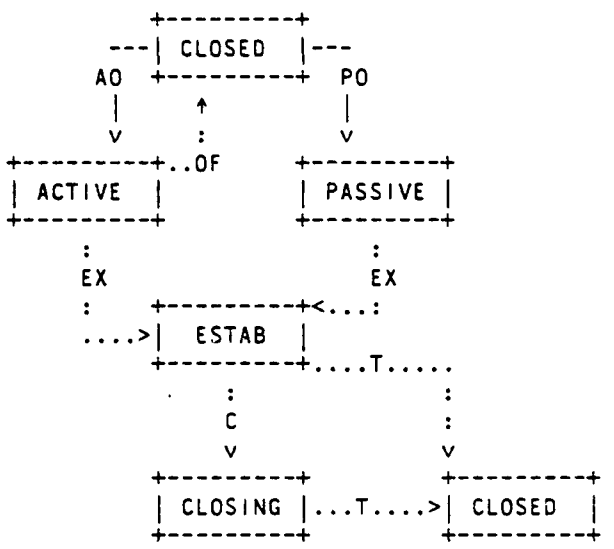
ULP Service Requests

- PO - passive open, either unspecified or fully specified
- AO - active open
- S - send
- C - close
- Ab - abort (forces to CLOSED)

TCP Internal Events

- EX - exchange of state information between local service entities
- T - termination of service due to service failure
- OF - open fail, active open request failed

Note: The first "actor" of the ULP-pair is defined to be ULP A, as shown by the notation PO/A, or AO/A.



TCP LOCAL SERVICE STATE MACHINE SUMMARY

LEGEND

Service Requests ---------->

- PO - passive open, either unspecified or fully specified
- AO - active open, with or without data
- S - send
- C - close
- A - abort (always leads to CLOSED state)

TCP Service Machine Internal Events .....>

- EX - exchange between service entities
- T - termination of service due to service failure

The preceding diagram summarizes the definition of the service state machine for the local service machine appearing in Section 3.2. This diagram presents the sequence of state changes from the point of view of a single ULP accessing TCP's services. The boxes represent the states of the state machine; the arrows represent state transitions resulting from the service requests and service responses shown. Please note that the diagram is intended only as a summary and does not supersede the formal definition of Section 3.2.

3.2.3 State Vector

The service machine vector of a local service machine consists of the following elements:

1. state = (CLOSED, ACTIVE OPEN, PASSIVE OPEN, ESTABLISHED, CLOSING);

2. source port - identifier of the local ULP.

3. source address - internet address identifying the location of the local ULP.

4. destination port - identifier of the remote ULP.

5. destination address - internet address identifying the location of the remote ULP.

6. local connection name - the shorthand identifier used in all service responses and service requests except for open requests.

7. original precedence - precedence level specified by the local ULP in the open request.

8. actual precedence - precedence level negotiated at connection opening and used during connection lifetime.

9. security - security information (including security level, compartment, handling restrictions, and transmission control code) defined by the local ULP.

10. ULP timeout - the longest delay allowed for data delivery before automatic connection termination.

11. open mode - the type of open request issued by the local ULP including UNPASSIVE, FULLPASSIVE, and ACTIVE.

12. send queue - storage location of data sent by the local ULP before transmission to the remote TCP. Each data octet is stored with a timestamp indicating its time of entry.

13. send queue length - number of entries in the recv queue made up of data and timestamp information.

14. send push - an offset from the front of the send queue indicating the end of push data.

15. send urgent - an offset from the front of the end queue indicating the end of urgent data.

16. receive queue - storage location of data received from the remote TCP before delivery to the local ULP.

17. receive queue length - number of data octets in the receive queue.

18. receive push - an offset from the front of the receive queue indicating the end of push data.

19. receive urgent - an offset from the front of the receive queue indicating the end of urgent data.

20. receive allocation - the number of data octets the local ULP is currently willing to receive.

A state machine's initial state is CLOSED with NULL values for all other state vector elements.

3.2.4 Data Structures

For clarity in the events and actions section, data structures are declared for the interaction primitives and their parameters. A subset of ADA data constructs, common to most high level languages, is used. However, a data structure may be partially typed or completely untyped where specific formats or data types are implementation dependent.

3.2.4.1 state vector The definition of the TCP service machine state vector appears in Section 3.2.3 above. The service machine state vectors for the two local TCP service machines are declared as:

```
sv_A : state_vector_type;

sv_B : state_vector_type;

type state_vector_type is
  record
     state : ( CLOSED, ACTIVE OPEN, PASSIVE OPEN,
               ESTABLISHED, CLOSING );
     source_addr : address_type;
     source_port : TWO_OCTETS;
     destination_addr : address_type;
     destination port : TWO_OCTETS;
     lcn : lcn_type;
     sec : security_type;
     original_prec : 0..7;
     actual_prec : 0..7;
     ulp_timeout : time_type;
```

```
        open_mode : (UNPASSIVE, FULLPASSIVE, ACTIVE);
        send_queue : timed_queue_type;
        send_queue_length : integer;
        send_push : integer;
        send_urg : integer;
        recv_queue : queue_type;
        recv_queue_length : integer;
        recv_push : integer;
        recv_urg : integer;
        recv_alloc : integer;
      end record;

    type timed_queue_type is queue (1..SIZE_OF_SEND_RESOURCE) of
        record
           data_octet : OCTET;
           timestamp  : time_type;
        end record;

    type queue_type is queue (1..SIZE_OF_RECV_RESOURCE) of
           data_octet : OCTET;
        end record;

    type address_type is FOUR_OCTETS;
    type lcn_type  : undefined;  --implementation dependent
    type time_type : undefined;  --implementation dependent

    subtype OCTET is INTEGER range 0..255;
    subtype TWO_OCTETS is INTEGER range 0..2**16-1
    subtype FOUR_OCTETS is INTEGER range 0..2**32-1;
```

3.2.4.2 from ULP The from_ULP structure holds the interface parameters and data associated with the service request primitives specified in Section 3.1.1 above. Although the structure is composed of the parameters from all the service requests, a particular service request will use only those structure elements corresponding to its specified parameters. This structure directly corresponds to the from_ULP structure declared in entity state machine specification, Section 6.3.4.2.

The from_ULP structure is declared as:

```
    type from_ULP_type is
      record
        request_name : (Unspecified_Passive_Open, Full_Passive_Open,
                        Active_Open, Active_Open_with_data,
                        Send, Allocate, Close, Abort, Status);
        source_addr
        source_port
        destination_addr
        destination_port
        lcn
        timeout
```

```
        precedence
        security
        data
        data_length
        push_flag
        urgent_flag
      end record;
```

3.2.4.3 to ULP The to_ULP structure holds interface parameters and data associated with the service response primitives, as specified in Section 3.1.2 above. Although the structure is composed of the parameters from all the service requests, a particular service response will use only those structure elements corresponding to its specified parameters. This structure directly corresponds to the to_ULP structure declared in Section 6.3.4.3 of the mechanism specification. The to_ULP structure is declared as:

```
type to_ULP_type is
  record
    service_response : (Open_Id, Open_Fail, Open_Success,
                        Deliver, Status_Response, Terminate,
                        Error);
    source_addr
    source_port
    destination_addr
    destination_port
    lcn
    data
    data_length
    urgent_flag
    error_desc
    status_block : status_block_type;
  end record;

type status_block_type is
  record
     connection_state
     send_window
     receive_window
     amount_of_unacked_data
     amount_of_unreceived_data
     urgent_state
     precedence
     security
     timeout
  end record;
```

3.2.5 Event List

The events for the TCP service machine are drawn form the service request primitives defined in Section 3.1.1 above. Optional service request parameters are shown in brackets. The capitalized list of parameters represent the actual values of the parameters passed by the service primitive. The event list:

1. Unspecified Passive Open(SOURCE_PORT,
   [,TIMEOUT] [,PRECEDENCE] [,SECURITY]);

2. Full Passive Open(SOURCE_PORT,
   DESTINATION_PORT, DESTINATION_ADDRESS,
   [,TIMEOUT] [,PRECEDENCE] [,SECURITY]);

3. Active Open(SOURCE_PORT,
   DESTINATION_PORT, DESTINATION_ADDRESS
   [,TIMEOUT] [,PRECEDENCE] [,SECURITY]);

4. Active Open w/data(SOURCE_PORT,
   DESTINATION_PORT, DESTINATION_ADDRESS
   [,TIMEOUT] [,PRECEDENCE] [,SECURITY]);
   DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG);

5. Send(LCN, DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG [,TIMEOUT]);

6. Allocate(LCN, DATA LENGTH)

7. Close(LCN)

8. Abort(LCN)

9. Status(LCN)

10. NULL - Although no service request is issued by a ULP, certain conditions within the TCP service machine produce a service response.

3.2.6 Events and Actions

For the purposes of this definition, the ULP and TCP entities are identified with the capitol letters "A" and "B". The first ULP to make a service request is labelled ULP "A"; its local service machine is TCP "A." The peer ULP and its TCP are labelled ULP B and TCP B. The service requests are labelled with the identifier of the issuing ULP, such as Close/A. The service responses are similarly labelled, such as Terminate/B.

A service request appearing with a "*" identifier may be issued by either ULP A or ULP B. The appropriate TCP handles the request updating its own state vector if necessary. The service response corresponding to such a request is directed to the appropriate ULP.

When a service request is invalid for the current state of the state machine, the service request appears without a parameter list.

In this service machine model, "simultaneous" service are treated as unordered sequential events. Hence, CLOSE/A occurring "simultaneously" with CLOSE/B is represented as occurring sequentially without intervening events. The order chosen for the event sequence should not alter the resulting state, so that a sequence such as (CLOSE/A, CLOSE/B) should lead to the same state as the (CLOSE/B, CLOSE/A) sequence.

The STATUS event produces the same service response from the TCP service machine in every state. Rather than show these in each state, the STATUS request and STATUS RESPONSE response are shown once here.

Event: STATUS(LCN)

```
Actions: STATUS_RESPONSE( LCN, SOURCE_PORT, SOURCE_ADDRESS,
                          DESTINATION_PORT, DESTINATION_ADDRESS,
                          PRECEDENCE, SECURITY, CONNECTION_STATE,
                          RECEIVE_WINDOW, SEND_WINDOW,
                          AMOUNT_WAITING_ACK, AMOUNT_WAITING_RECEIPT,
                          URGENT_MODE, TIMEOUT );
```

3.2.6.1 Event/Actions Specifications The following section is organized by composite state. Mirror-image composite states, such as PASSIVE/ACTIVE and ACTIVE/PASSIVE, appear as just one. Only one-way data transfer is represented by the service machine. since the data transfer service is symmetric. Thus, a definition of bi-directional data transfer can be provided by duplicating the existing one-way definition.

Certain conditions checks and groups of actions occur in several places and have been formed into decision functions and action procedures. The decisions function definitions appear in Section 3.2.6.2. The action procedure definitions appear in Section 3.2.6.3.

```
==================
 STATE_A = CLOSED
 STATE_B = CLOSED
==================

Event: Unspecified Passive Open/A( SOURCE_PORT [,TIMEOUT]
                                   [,PRECEDENCE] [,SECURITY] )

Actions:  record_open_parameters(A, UNPASSIVE);
          sv_A.lcn := assign_new_lcn;
          open_id( sv_A.lcn, sv_a.source_port, sv_A.source_addr, NULL, NULL );
          TRANSFER to_ULP to the ULP named by sv_A.source_port;
          sv_A.state := PASSIVE_OPEN;


Event: Full Passive Open/A( SOURCE_PORT,
                            DESTINATION_PORT, DESTINATION_ADDRESS
                            [,TIMEOUT] [,PRECEDENCE] [,SECURITY] )

Actions:  record_open_parameters(A, FULLPASSIVE);
          sv_A.lcn := assign_new_lcn;
          open_id( sv_A.lcn, sv_A.source_port, sv_A.source_addr,
                             sv_A.destination_port, sv_A.destination_addr);
          TRANSFER to_ULP to the ULP named by sv_A.source_port;
          sv_A.state := PASSIVE_OPEN;


Event: Active Open/A( SOURCE_PORT, DESTINATION_PORT, DESTINATION_ADDRESS
                      [,TIMEOUT] [,PRECEDENCE] [,SECURITY] )

Actions:  record_open_parameters(A, ACTIVE);
          sv_A.lcn := assign_new_lcn;
          open_id( sv_A.lcn, sv_A.source_port, sv_A.source_addr,
                             sv_A.destination_port, sv_A.destination_addr);
          TRANSFER to_ULP to the ULP named by sv_A.source_port;
          sv_A.state := ACTIVE_OPEN;


Event: Active Open with data/A( SOURCE_PORT,
                                DESTINATION_PORT, DESTINATION_ADDRESS
                                [,TIMEOUT] [,PRECEDENCE] [,SECURITY]
                                DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG )

Actions: sv_A.lcn := assign_new_lcn;
          open_id( sv_A.lcn, sv_A.source_port, sv_A.source_addr,
                             sv_A.destination_port, sv_A.destination_addr);
         TRANSFER to_ULP to the ULP named by sv_A.source_port;
         if (room_in(sv_A.send_queue)
         then
              add_to_send_queue(sv_A);
              record_open_parameters(A, ACTIVE);
              sv_A.state := ACTIVE_OPEN;
```

```
        else
             openfail( sv_A.lcn );
             TRANSFER to_ULP to the ULP named by sv_A.source_port;


Event: Close/A( LCN )
       or Abort/A( LCN )
       or Allocate/A( LCN, DATA_LENGTH );

Actions: error(sv_A.lcn, "Connection does not exist.");
         TRANSFER to_ULP to the ULP named by sv_A.source_port;
```

```
=======================
 STATE_A = PASSIVE OPEN
 STATE_B = CLOSED
=======================

Event: Close/A( LCN )
       or Abort/A( LCN )

Actions: initialize( sv_A );
         sv_A.state := CLOSED;

Event: Unspecified Passive Open/B( SOURCE_PORT [,TIMEOUT]
                                   [,PRECEDENCE] [,SECURITY] )

Actions:  sv_B.lcn := assign_new_lcn;
          record_open_parameters(B, UNPASSIVE);
          open_id( sv_B.lcn, sv_B.source_port, sv_B.source_addr, NULL, NULL );
          TRANSFER to_ULP to the ULP named by sv_B.source_port;
          sv_B.state := PASSIVE_OPEN;



Event: Full Passive Open/B( SOURCE_PORT,
                            DESTINATION_PORT, DESTINATION_ADDRESS
                            [,TIMEOUT] [,PRECEDENCE] [,SECURITY] )

Actions:  sv_B.lcn := assign_new_lcn;
          record_open_parameters(B, FULLPASSIVE);
          open_id( sv_B.lcn, sv_B.source_port, sv_B.source_addr,
                             sv_B.destination_port, sv_B.destination_addr );
          TRANSFER to_ULP to the ULP named by sv_B.source_port;
          sv_B.state := PASSIVE_OPEN;


Event: Active Open/B( SOURCE_PORT, DESTINATION_PORT, DESTINATION_ADDRESS
                 [,TIMEOUT] [,PRECEDENCE] [,SECURITY] )

Actions:  record_open_parameters(B, ACTIVE);
          sv_B.lcn := assign_new_lcn;
          open_id( sv_B.lcn, sv_B.source_port, sv_B.source_addr,
                             sv_B.destination_port, sv_B.destination_addr );
          TRANSFER to_ULP to the ULP named by sv_B.source_port;
          sv_B.state := ACTIVE_OPEN;


Event: Active Open with data/B( SOURCE_PORT,
                                DESTINATION_PORT, DESTINATION_ADDRESS
                                [,TIMEOUT] [,PRECEDENCE] [,SECURITY]
                                DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG )

Actions: sv_B.lcn := assign_new_lcn;
          open_id( sv_B.lcn, sv_B.source_port, sv_B.source_addr,
```

```
                        sv_B.destination_port, sv_B.destination_addr );
        TRANSFER to_ULP to the ULP named by sv_B.source_port;
        if (room_in(sv_B.send_queue)
        then
             add_to_send_queue(sv_B);
             record_open_parameters(B, ACTIVE);
             sv_B.state := ACTIVE_OPEN;
        else
             openfail( sv_B.lcn );
             TRANSFER to_ULP to the ULP named by sv_B.source_port;

Event: Allocate/A( LCN, DATA_LENGTH )

Actions: sv_A.recv_alloc := sv_A.recv_alloc + DATA_LENGTH;


Event: Full Passive Open/A( )
       or Send/A( )

Actions: error(sv_A.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_A.source_port;


Event: Close/B( )
       or Abort/B( )
       or Send/B( )
       or Allocate/B( )

Actions: error(sv_B.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;
```

```
======================
 STATE_A = ACTIVE OPEN
 STATE_B = CLOSED
======================

Event: Close/A( LCN )
       or Abort/A( LCN )

Actions: initialize( sv_A );
         sv_A.state := CLOSED;


Event: Allocate/A( LCN, DATA_LENGTH )

Actions: sv_A.recv_alloc := sv_A.recv_alloc + DATA_LENGTH;


Event: Send/A( LCN, DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG
               [,TIMEOUT] )

Actions: if (room_in(sv_A.send_queue))
         then if (TIMEOUT !!= NULL)
              then sv_A.ulp_timeout := TIMEOUT;
              add_to_send_queue(sv_A);
         else error(sv_A.lcn, "insufficient resources.");
              TRANSFER to_ULP to the ULP named by sv_A.source_port;


Event: Unspecified Passive Open/B( SOURCE_PORT [,TIMEOUT]
                                   [,PRECEDENCE] [,SECURITY] )

Actions:  sv_B.lcn := assign_new_lcn;
          record_open_parameters(B, UNPASSIVE);
          open_id( sv_B.lcn, sv_B.source_port, sv_B.source_addr, NULL, NULL );
          TRANSFER to_ULP to the ULP named by sv_B.source_port;
          sv_B.state := PASSIVE_OPEN;


Event: Full Passive Open/B( SOURCE_PORT,
                            DESTINATION_PORT, DESTINATION_ADDRESS
                            [,TIMEOUT] [,PRECEDENCE] [,SECURITY] )

Actions:  sv_B.lcn := assign_new_lcn;
          record_open_parameters(B, FULLPASSIVE);
          open_id( sv_B.lcn, sv_B.source_port, sv_B.source_addr,
                             sv_B.destination_port, sv_B.destination_addr );
          TRANSFER to_ULP to the ULP named by sv_B.source_port;
          sv_B.state := PASSIVE_OPEN;


Event: Active Open/B( SOURCE_PORT,
                      DESTINATION_PORT, DESTINATION_ADDRESS
```

```
                    [,TIMEOUT] [,PRECEDENCE] [,SECURITY] )

Actions:  sv_B.lcn := assign_new_lcn;
          record_open_parameters(B, ACTIVE);
          open_id( sv_B.lcn, sv_B.source_port, sv_B.source_addr,
                             sv_B.destination_port, sv_B.destination_addr );
          TRANSFER to_ULP to the ULP named by sv_B.source_port;
          sv_B.state := ACTIVE_OPEN;


Event: Active Open with data/B( SOURCE_PORT,
                                DESTINATION_PORT, DESTINATION_ADDRESS
                                [,TIMEOUT] [,PRECEDENCE] [,SECURITY]
                                DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG )

Actions: sv_B.lcn := assign_new_lcn;
         open_id( sv_B.lcn, sv_B.source_port, sv_B.source_addr,
                            sv_B.destination_port, sv_B.destination_addr );
         TRANSFER to_ULP to the ULP named by sv_B.source_port;
         if (room_in(sv_B.send_queue)
         then add_to_send_queue(sv_B);
              record_open_parameters(B, ACTIVE);
              sv_B.state := ACTIVE_OPEN;

         else openfail( sv_B.lcn );
              TRANSFER to_ULP to the ULP named by sv_B.source_port;


Event: Full Passive Open/A( )
       or Active Open/A( )
       or Active Open with data/A( )

Actions: error(sv_A.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_A.source_port;


Event: Send/B( )
       or Close/B( )
       or Abort/B( )

Actions: error(sv_B.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;

Event: NULL

Actions:  Internal Events

  1) if timeout_exceeded(sv_A)
     then openfail( sv_A.lcn);
          TRANSFER to_ULP to the ULP named by sv_A.source_port;
          initialize( sv_A );
          sv_A.state := CLOSED;
```

```
=======================
 STATE_A = PASSIVE OPEN
 STATE_B = ACTIVE OPEN
=======================

Event: Close/*( LCN )
       or Abort/*( LCN )

Actions: initialize( sv_* );
         sv_*.state := CLOSED;


Event: Allocate/*( LCN, DATA_LENGTH )

Actions: sv_*.recv_alloc := sv_*.recv_alloc + DATA_LENGTH;


Event: Send/B( LCN, DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG
               [,TIMEOUT] )

Actions: if (room_in(sv_B.send_queue)
         then
              add_to_send_queue(sv_B);
              if (TIMEOUT !!= NULL)
              then sv_B.ulp_timeout := TIMEOUT;
         else
              error( sv_B.lcn, "Insufficient resources.");
              TRANSFER to_ULP to the ULP named by sv_B.source_port;


Event: Send/A( )

Actions: error(sv_A.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_A.source_port;


Event: Full Passive Open/*( )
       or ActiveOpen/*( )
       or ActiveOpen with data/*( )

Actions: error(sv_*.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_*.source_port;


Event: NULL

Actions: Internal Events

  1) if (sv_A.sec !!= sv_B.sec)
     then
          openfail( sv_B.lcn );
          TRANSFER to_ULP to the ULP named by sv_B.source_port;
```

```
          initialize( sv_B );
          sv_B.state := CLOSED;

     else --Take greater precedence level to model precedence negotiation;
          --If negotiation is not supported, mismatched precedence
          --is handled the same as mismatched security.
          if (sv_A.original_prec !!= sv_B.original_prec)
          then
              sv_A.actual_prec := maximum(sv_A.original_prec,
                                          sv_B.original_prec);
              sv_B.actual_prec := maximum(sv_A.original_prec,
                                          sv_B.original_prec);
          if (sv_A.open_mode = UNPASSIVE)
          then
               sv_A.destination_addr := sv_B.source_addr;
               sv_A.destination_port := sv_B.source_port;

          sv_A.state := ESTABLISHED;
          open_success( sv_A.lcn );
          TRANSFER to_ULP to the ULP named by sv_A.source_port;
          sv_B.state := ESTABLISHED;
          open_success( sv_B.lcn );
          TRANSFER to_ULP to the ULP named by sv_B.source_port;

OR,
  2) if timeout_exceeded(sv_B)
     then
          openfail( sv_B.lcn );
          TRANSFER to_ULP to the ULP named by sv_B.lcn;
          initialize( sv_B );
          sv_B.state := CLOSED;
```

```
============================
STATE_A = PASSIVE OPEN
STATE_B = PASSIVE OPEN
============================

Event: Allocate/*( LCN, DATA_LENGTH )

Actions: sv_*.recv_alloc := sv_*.recv_alloc + DATA_LENGTH;


Event: Close/*( LCN )
       or Abort/*( LCN )

Actions: initialize( sv_* );
         sv_*.state := CLOSED


Event: Full Passive Open/*( )
       or ActiveOpen/*( )
       or ActiveOpen with data/*( )
       or Send/*( )

Actions: error(sv_*.lcn, "illegal request.");
         TRANSFER to_ULP to the ULP named by sv_*.source_port;
```

```
=======================
 STATE_A = ACTIVE OPEN
 STATE_B = ACTIVE OPEN
=======================

Event: Allocate/*( LCN, DATA_LENGTH )

Actions: sv_*.recv_alloc := sv_*.recv_alloc + DATA_LENGTH;


Event: Close/*( LCN )
       or Abort/*( LCN )

Actions: initialize( sv_* );
         sv_*.state := CLOSED;


Event: Full Passive Open/*( )
       or Active Open/*( )
       or Active Open with data/*( )

Actions: error(sv_*.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_*.source_port;


Event: Send/*( LCN, DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG
               [,TIMEOUT] )

Actions: if (room_in(sv_*.send_queue)
         then
              add_to_send_queue(sv_*);
              if (TIMEOUT !!= NULL)
              then sv_*.ulp_timeout := TIMEOUT;
         else
              error( sv_*.lcn, "Insufficient resources.");
              TRANSFER to_ULP to the ULP named by sv_*.source_port;


Event: NULL

Actions: Internal Events

  1) if (sv_A.sec !!= sv_B.sec)
     then
          openfail( sv_A.lcn );
          Transfer to_ULP to the ULP named by sv_A.source_port;
          openfail( sv_B.lcn );
          Transfer to_ULP to the ULP named by sv_B.source_port;
          initialize( sv_A );      sv_A.state := CLOSED;
          initialize( sv_B );      sv_B.state := CLOSED;

     else --take greater precedence level to model precedence negotiation;
```

```
        --if negotiation not supported, mismatched precedence
        --is handled just as mismatched security
        if (sv_A.original_prec !!= sv_B.original_prec)
        then
            sv_A.actual_prec := maximum(sv_A.original_prec,
                                        sv_B.original_prec);
            sv_B.actual_prec := maximum(sv_A.original_prec,
                                        sv_B.original_prec);
        sv_A.state := ESTABLISHED;
        sv_B.state := ESTABLISHED;
        open_success( sv_A.lcn );
        TRANSFER to_ULP to the ULP named by sv_A.source_port;
        open_success( sv_B.lcn );
        TRANSFER to_ULP to the ULP named by sv_A.source_port;

OR,
  2) if timeout_exceeded(sv_A)
     then
         openfail( sv_A.lcn );
         TRANSFER to_ULP to the ULP named by sv_A.source_port;
         initialize(sv_A);
         sv_A.state := CLOSED;
OR,
  3) if timeout_exceeded(sv_A)
     then
         openfail( sv_B.lcn );
         TRANSFER to_ULP to the ULP named by sv_B.source_port;
         initialize( sv_B );
         sv_B.state := CLOSED;
```

```
======================
 STATE_A = ESTABLISHED
 STATE_B = ESTABLISHED
======================

Event: Send/*( LCN, DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG
               [,TIMEOUT] )

Actions: if (room_in(sv_*.send_queue)
             add_to_send_queue(sv_*);
             if (TIMEOUT !!= NULL)
             then sv_*.ulp_timeout := TIMEOUT;
         else
             error( sv_*.lcn, "Insufficient resources.");
             TRANSFER to_ULP to the ULP named by sv_*.source_port;


Event: Allocate/*( LCN, DATA_LENGTH );

Actions: sv_*.recv_alloc := sv_*.recv_alloc + DATA_LENGTH;
         if (sv_*.recv_queue_length > 0)
         then try_to_deliver;


Event: Abort/*( LCN )

Actions: terminate(sv_*.lcn, "User abort.");
         TRANSFER to_ULP to the ULP named by sv_*.source_port;
         initialize( sv_* );
         sv_*.state := CLOSED;


Event: Close/*( LCN )

Actions:  sv_*.send_push := sv_*.send_queue_length;
          sv_*.state := CLOSING;


Event: Full Passive Open/*( )
       or Active Open/*( )
       or Active Open with data/*( )

Actions: error(sv_*.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_*.source_port;


Event: NULL

Actions: Internal Events

   --For clarity, one-way data transport, from TCP A to TCP B is shown.
   --Because the data transport service is symmetric, the following
```

```
   --text could be duplicated to represent bi-directional data transport.

  1) if timeout_exceeded(sv_A)
     then
          terminate(sv_A.lcn, "ULP timeout.");
          TRANSFER to_ULP to the ULP named by sv_A.source_port;
          initialize(sv_A);
          sv_A.state := CLOSED;

 OR,
  2) if (conditions exist such that no data can be exchanged
          by local state machines )
     then
          terminate(sv_A.lcn, "Service failure.");
          TRANSFER to_ULP to the ULP named by sv_A.source_port;
          terminate(sv_B.lcn, "Service failure.");
          TRANSFER to_ULP to the ULP named by sv_B.source_port;
          initialize(sv_A);     sv_A.state := CLOSED;
          initialize(sv_B);     sv_B.state := CLOSED;

OR,
  3) if (the data exchange between local state machines is triggered)
     then
          if (sv_A.send_urg !!= 0)
          then
            sv_B.recv_urg := (sv_B.recv_queue_length + sv_A.send_urg);

          Dequeue some portion of data equal to "amount"
          (amount may be >= 0) from sv_A.send_queue
          and append to sv_B.recv_queue;

          if (amount > 0)
          then
               sv_A.send_queue_length := sv_A.send_queue_length - amount;
               sv_B.recv_queue_length := sv_B.recv_queue_length + amount;

               if (sv_A.send_urg =< amount)
               then sv_A.send_urg := 0;
               else sv_A.send_urg := sv_A.send_urg - amount;

               if (sv_A.send_push =< amount)
               then sv_B.recv_push := sv_B.recv_push + sv_A.send_push;
                    sv_A.send_push := 0;
               else sv_A.send_push := sv_A.send_push - amount;
               try_to_deliver;
```

```
======================
 STATE_A = ESTABLISHED
 STATE_B = CLOSING
======================


Event: Send/A( LCN, DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG
               [,TIMEOUT] )

Actions: if (room_in(sv_A.send_queue)
             add_to_send_queue(sv_A);
             if (TIMEOUT !!= NULL)
             then sv_A.ulp_timeout := TIMEOUT;
         else
             error( sv_A.lcn, "Insufficient resources.");
             TRANSFER to_ULP to the ULP named by sv_A.source_port;


Event: Close/A( LCN )

Actions:  sv_A.send_push := sv_A.send_queue_length;
          sv_A.state  := CLOSING;


Event: Allocate/*( LCN, DATA_LENGTH );

Actions: sv_*.recv_alloc := sv_*.recv_alloc + DATA_LENGTH;
         if (sv_*.recv_queue_length > 0)
         then try_to_deliver;


Event:  Abort/*( LCN )

Actions: initialize( sv_* );
         sv_*.state := CLOSED;
         terminate(sv_*.lcn, "User abort.");
         TRANSFER to_ULP to the ULP named by sv_*.source_port;


Event: Send/B( )
       or Close/B( )

Actions: error(sv_B.lcn, "Connection closing.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;


Event: Active Open/*( )
       or Active Open with data/*( )
       or Full Passive Open/*( )

Actions: error(sv_*.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_*.source_port;
```

```
Event: NULL

Actions: Internal Events

  1) if timeout_exceeded(sv_*)
     then
          terminate(sv_*.lcn, "ULP timeout.");
          TRANSFER to_ULP to the ULP named by sv_*.source_port;
          initialize(sv_*);
          sv_*.state := CLOSED;

OR,
  2) if (conditions exist such that no data can be exchanged
          by local state machines )
     then
          terminate(sv_A.lcn, "Service failure.");
          TRANSFER to_ULP to the ULP named by sv_A.source_port;
          terminate(sv_B.lcn, "Service failure.");
          TRANSFER to_ULP to the ULP named by sv_B.source_port;
          initialize(sv_A);    sv_A.state := CLOSED;
          initialize(sv_B);    sv_B.state := CLOSED;

OR,
  3) if (contents sv_B.send_queue have all been transferred
        to sv_A.recv_queue and subsequently delivered to ULP A )
     then
          closing( sv_A.lcn );
          TRANSFER to_ULP to the ULP named by sv_A.source_port;

OR,
  4) --For clarity, one-way data transport, from TCP A to TCP B is shown.
     --Because the data transport service is symmetric, the following
     --text could be duplicated to represent bi-directional data transport.
     --Note that TCP B is still responsible to reliably transport any
     --data remaining in sv_B.send_queue.

     if (the data exchange between local state machines has been triggered)
     then
          if (sv_A.send_urg !!= 0)
          then
             sv_B.recv_urg equal := (sv_B.recv_queue_length + sv_A.send_urg);

          Dequeue some portion of data equal to "amount"
          (amount may be >= 0) from sv_A.send_queue
          and append to sv_B.recv_queue;

          if (amount > 0)
          then
               sv_A.send_queue_length := sv_A.send_queue_length - amount;
               sv_B.recv_queue_length := sv_B.recv_queue_length + amount;

               if (sv_A.send_urg =< amount)
```

```
then sv_A.send_urg := 0;
else sv_A.send_urg := sv_A.send_urg - amount;

if (sv_A.send_push =< amount)
then sv_B.recv_push := sv_B.recv_push + amount;
     sv_A.send_push := 0;
else sv_A.send_push := sv_A.send_push - amount;
try_to_deliver;
```

```
==================
 STATE_A = CLOSING
 STATE_B = CLOSING
==================

Event:  Abort/*( LCN )

Actions: initialize(sv_*);
         sv_*.state := CLOSED;
         terminate(sv_*.lcn, "User Abort.");
         TRANSFER to_ULP to the ULP named by sv_*.source_port;


Event: Allocate/*( LCN, DATA_LENGTH );

Actions: sv_*.recv_alloc := sv_*.recv_alloc + DATA_LENGTH;
         if (sv_*.recv_queue_length > 0)
         then try_to_deliver;


Event: Send/*( )
       or Close/*( )

Actions: error(sv_*.lcn, "Connection closing.");
         TRANSFER to_ULP to the ULP named by sv_*.source_port;


Event: Active Open/*( )
       or Active Open with data/*( )
       or Full Passive Open/*( )

Actions: error(sv_*.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_*.source_port;


Event: NULL

Actions: Internal Events

  1)--For clarity, one-way data transport, from TCP A to TCP B is shown.
    --Because the data transport service is symmetric, the following
    --text could be duplicated to represent bi-directional data transport.
    --Note that TCP B is still responsible to reliably transport any
    --data remaining in sv_B.send_queue.

    if (the data exchange between local state machines has been triggered)
    then
         if (sv_A.send_urg != 0)
         then
            sv_B.recv_urg equal := (sv_B.recv_queue_length + sv_A.send_urg);

         Dequeue some portion of data equal to "amount"
```

```
          (amount may be >= 0) from sv_A.send_queue
          and append to sv_B.recv_queue;

          if (amount > 0)
          then
               sv_A.send_queue_length := sv_A.send_queue_length - amount;
               sv_B.recv_queue_length := sv_B.recv_queue_length + amount;

               if (sv_A.send_urg =< amount)
               then sv_A.send_urg := 0;
               else sv_A.send_urg := sv_A.send_urg - amount;

               if (sv_A.send_push =< amount)
               then sv_B.recv_push := sv_B.recv_push + amount;
                    sv_A.send_push := 0;
               else sv_A.send_push := sv_A.send_push - amount;
               try_to_deliver;

OR,
  2) if ((contents sv_B.send_queue have all been transferred
            to sv_A.recv_queue and subsequently delivered to ULP A )
        &
         (contents sv_A.send_queue have all been transferred
            to sv_B.recv_queue and subsequently delivered to ULP B ))

     then
          terminate(sv_A.lcn, "Connection closed.");
          TRANSFER to_ULP to the ULP named by sv_A.source_port;
          terminate(sv_B.lcn, "Connection closed.");
          TRANSFER to_ULP to the ULP named by sv_B.source_port;
          initialize(sv_A);        sv_A.state := CLOSED;
          initialize(sv_B);        sv_B.state := CLOSED;

OR,
  3) if timeout_exceeded(sv_*)
     then
          terminate(sv_*.lcn, "ULP timeout.");
          TRANSFER to_ULP to the ULP named by sv_*.source_port;
          initialize(sv_*);
          sv_*.state := CLOSED;
```

```
--The composite states, CLOSED/ESTABLISHED, CLOSING/ESTABLISHED,
--ACTIVE/ESTABLISHED, ACTIVE/CLOSING, PASSIVE/ESTABLISHED, AND
--PASSIVE/CLOSING, are reached after abnormal
--connection termination caused by either an Abort request or
--service failure. Because the service request lists for ULP A
--already appear in other states, these lists are referenced rather
--than duplicated.
```

```
=======================
 STATE_A = CLOSED
 STATE_B = ESTABLISHED
=======================

--ULP A's service request list appears in the CLOSED/CLOSED state.

Event: Send/B( LCN, DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG
               [,TIMEOUT] )

Actions: if (room_in(sv_B.send_queue))
             if (TIMEOUT !!= NULL)
             then sv_B.ulp_timeout := TIMEOUT;
             add_to_send_queue(sv_B);
         else
             error( sv_B.lcn, "Insufficient resources.");
             TRANSFER to_ULP to the ULP named by sv_B.source_port;


Event: Allocate/B( LCN, DATA_LENGTH );

Actions: sv_B.recv_alloc := sv_B.recv_alloc + DATA_LENGTH;
         if (sv_B.recv_queue_length > 0)
         then try_to_deliver;


Event: Close/B( LCN )

Actions: sv_B.push := sv_B.send_queue_length;
         sv_B.state := CLOSING;


Event: Abort/B( LCN )

Actions: terminate(sv_B.lcn, "User abort.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;
         initialize(sv_B);
         sv_B.state := CLOSED;


Event: Full Passive Open/B( )
       Active Open/B( )
       Active Open with Data/B( )

Actions: error(sv_B.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;


Event: NULL

Actions: Internal Events
```

```
  1) terminate(sv_B.lcn, "Remote Abort.");
     TRANSFER to_ULP to the ULP named by sv_B.source_port;
     initialize(sv_B);
     sv_B.state := CLOSED;

OR,
  2) if timeout_exceeded(sv_B)
     then
         terminate(sv_B.lcn, "User timeout.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;
         initialize(sv_B);
         sv_B.state := CLOSED;
```

```
======================
 STATE_A = CLOSED
 STATE_B = CLOSING
======================

--ULP A's service request list appears in the CLOSED/CLOSED state.

Event: Abort/B( LCN )

Actions: terminate(sv_B.lcn, "User Abort.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;
         initialize(sv_B);
         sv_B.state := CLOSED;


Event: Allocate/B( LCN, DATA_LENGTH );

Actions: sv_B.recv_alloc := sv_B.recv_alloc + DATA_LENGTH;
         if (sv_B.recv_queue_length > 0) then try_to_deliver;


Event: Close/B( LCN )
       or Send/B( LCN )

Actions: error( sv_B.lcn, "Connection closing.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;


Event: Full Passive Open/B( LCN )
       Active Open/B( LCN )
       Active Open with Data/B( LCN )

Actions: error( sv_B.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;


Event: NULL

Actions: Internal Events

  1) terminate(sv_B.lcn, "Remote Abort.");
     TRANSFER to_ULP to the ULP named by sv_B.source_port;
     initialize(sv_B);
     sv_B.state := CLOSED;

OR,
  2) if timeout_exceeded(sv_B)
     then
         terminate(sv_B.lcn, "User timeout.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;
         initialize( sv_B );
         sv_B.state := CLOSED;
```

```
======================
 STATE_A = ACTIVE
 STATE_B = ESTABLISHED
======================

--ULP A's service request list appears in the ACTIVE/CLOSED state.

Event: Send/B( LCN, DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG
               [,TIMEOUT] )

Actions: if (room_in(sv_B.send_queue)
             if (TIMEOUT !!= NULL)
             then sv_B.ulp_timeout := TIMEOUT;
             add_to_send_queue(sv_B);
         else
             error( sv_B.lcn, "Insufficient resources.");
             TRANSFER to_ULP to the ULP named by sv_B.source_port;


Event: Close/B( LCN )

Actions: sv_B.push := sv_B.send_queue_length;
         sv_B.state := CLOSING;


Event: Abort/B( LCN )

Actions: terminate(sv_B.lcn, "User abort.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;
         initialize(sv_B);
         sv_B.state := CLOSED;


Event: Allocate/B( LCN, DATA_LENGTH );

Actions: sv_B.recv_alloc := sv_B.recv_alloc + DATA_LENGTH;
         if (sv_B.recv_queue_length > 0)
         then try_to_deliver;


Event: Full Passive Open/B( )
       Active Open/B( )
       Active Open with Data/B( )

Actions: error(sv_B.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;


Event: NULL

Actions: Internal Events
```

```
  1) terminate(sv_B.lcn, "Remote Abort.");
     TRANSFER to_ULP to the ULP named by sv_B.source_port;
     initialize(sv_B);
     sv_B.state := CLOSED;

OR,
  2) if timeout_exceeded(sv_B)
     then
          terminate(sv_B.lcn, "User timeout.");
          TRANSFER to_ULP to the ULP named by sv_B.source_port;
          initialize(sv_B);
          sv_B.state := CLOSED;

OR,
  3) if timeout_exceeded(sv_A)
     then
          terminate(sv_A.lcn, "User timeout.");
          TRANSFER to_ULP to the ULP named by sv_A.source_port;
          initialize(sv_A);
          sv_A.state := CLOSED;
```

```
======================
 STATE_A = ACTIVE
 STATE_B = CLOSING
======================

--ULP A's service request list appears in the ACTIVE/CLOSED state.

Event: Send/B( )
       or Close/B( )

Actions: error(sv_B.lcn, "Connection closing.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;


Event: Allocate/B( LCN, DATA_LENGTH );

Actions: sv_B.recv_alloc := sv_B.recv_alloc + DATA_LENGTH;
         if (sv_B.recv_queue_length > 0)
         then try_to_deliver;


Event: Abort/B( LCN )

Actions: terminate(sv_B.lcn, "User abort.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;
         initialize(sv_B);
         sv_B.state := CLOSED;


Event: Full Passive Open/B( )
       or Active Open/B( )
       or Active Open with Data/B( )

Actions: error(sv_B.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;


Event: NULL

Actions: Internal Events

  1) terminate(sv_B.lcn, "Remote Abort.");
     TRANSFER to_ULP to the ULP named by sv_B.source_port;
     initialize(sv_B);
     sv_B.state := CLOSED;

OR,
  2) if timeout_exceeded(sv_B)
     then
          terminate(sv_B.lcn, "User timeout.");
          TRANSFER to_ULP to the ULP named by sv_B.source_port;
          initialize(sv_B);
```

```
          sv_B.state := CLOSED;

OR,
  3) if timeout_exceeded(sv_A)
     then
          terminate(sv_A.lcn, "User timeout.");
          TRANSFER to_ULP to the ULP named by sv_A.source_port;
          initialize(sv_A);
          sv_A.state := CLOSED;
```

```
======================
 STATE_A = PASSIVE
 STATE_B = ESTABLISHED
======================

--ULP A's request list appears in the PASSIVE/CLOSED state.

Event: Send/B( LCN, DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG
               [,TIMEOUT] )

Actions: if (room_in(sv_B.send_queue)
             if (TIMEOUT !!= NULL)
             then sv_B.ulp_timeout := TIMEOUT;
             add_to_send_queue(sv_B);
         else
             error( sv_B.lcn, "Insufficient resources.");
             TRANSFER to_ULP to the ULP named by sv_B.source_port;


Event: Close/B( LCN )

Actions: sv_B.push := sv_B.send_queue_length;
         sv_B.state := CLOSING;


Event: Abort/B( LCN )

Actions: terminate(sv_B.lcn, "User abort.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;
         initialize(sv_B);
         sv_B.state := CLOSED;


Event: Allocate/B( LCN, DATA_LENGTH );

Actions: sv_B.recv_alloc := sv_B.recv_alloc + DATA_LENGTH;
         if (sv_B.recv_queue_length > 0)
         then try_to_deliver;


Event: NULL

Actions: Internal Events

  1) terminate(sv_B.lcn, "Remote Abort.");
     TRANSFER to_ULP to the ULP named by sv_B.source_port;
     initialize(sv_B);
     sv_B.state := CLOSED;

OR,
  2) if timeout_exceeded(sv_B)
     then
```

```
terminate(sv_B.lcn, "User timeout.");
TRANSFER to_ULP to the ULP named by sv_B.source_port;
initialize(sv_B);
sv_B.state := CLOSED;
```

```
=======================
 STATE_A = PASSIVE
 STATE_B = CLOSING
=======================

--ULP A's request list appears in the PASSIVE/CLOSED state.

Event: Allocate/B( LCN, DATA_LENGTH );

Actions: sv_B.recv_alloc := sv_B.recv_alloc + DATA_LENGTH;
         if (sv_B.recv_queue_length > 0)
         then try_to_deliver;


Event: Abort/B( LCN )

Actions: terminate(sv_B.lcn, "User abort.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;
         initialize(sv_B);
         sv_B.state := CLOSED;


Event: Close/B( LCN )
       or Send/B( LCN )

Actions: error( sv_B.lcn, "Connection closing.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;


Event: Full Passive Open/B( LCN )
       or Active Open/B( LCN )
       or Active Open with Data/B( LCN )

Actions: error( sv_B.lcn, "Illegal request.");
         TRANSFER to_ULP to the ULP named by sv_B.source_port;


Event: NULL

Actions: Internal Events

  1) terminate(sv_B.lcn, "Remote Abort.");
     TRANSFER to_ULP to the ULP named by sv_B.source_port;
     initialize(sv_B);
     sv_B.state := CLOSED;

OR,
  2) if timeout_exceeded(sv_B)
     then terminate(sv_B.lcn, "User timeout.");
          TRANSFER to_ULP to the ULP named by sv_B.source_port;
          initialize(sv_B);
          sv_B.state := CLOSED;
```

3.2.6.2 Decision Functions The decision functions represent condition checks made in several places in the service state machine definition.

3.2.6.2.1 room in(state vector name); The room_in decision function compares the amount of space available in the send_queue of the state_vector (named by the parameter) against the amount of data provided by the ULP in an Active Open with Data or a Send service request.
The data effects of this function are:

- Data examined:
  from_ULP.data_length SIZE_OF_SEND_RESOURCES
  sv_*.send_queue_length

- Return values:
  FALSE - The send_queue cannot accommodate all the data provided in the service request.
  TRUE - There is enough room in the send_queue for the data.

```
if (from_ULP.data_length >
            ( SIZE_OF_SEND_RESOURCE - sv_*.send_queue_length)
then return( FALSE )
else return( TRUE );
```

3.2.6.2.2 timeout exceeded(sv_*); The timeout_exceeded decision function compares the current time against the age of the data in the send_queue and the specified ULP timeout limit to determine if the ULP timeout has been exceeded.
The data effects of this function are:

- Data examined: sv_*.ulp_timeout sv_*.send_queue

- Return values:
  FALSE - The data in the send_queue does not exceed the ULP defined timeout limit.
  TRUE - Data in the send_queue has exceeded the timeout limit.

```
--The data at the front of the queue is the oldest.

if (sv_*.send_queue_length > 0)
then if ( CURRENT_TIME > sv_*.send_queue[0] + sv_*.ulp_timeout )
     then return(TRUE)
     else return(FALSE);
```

3.2.6.3 Action Procedures These routines appear in several places in the service machine definition. The "*" can be replace by either A or B for delivery to the appropriate ULP.

3.2.6.3.1 add to send queue(sv_*) The add_to_send_queue actions procedure enqueues the data provided in an Active Open with Data or Send request onto the send_queue of the state vector named by parameter.
The data effects of this procedure are:

- Data examined:
  from_ULP.data from_ULP.urgent_flag
  from_ULP.data_length from_ULP.push_flag

- Data modified:
  sv_*.send_queue sv_*.send_push
  sv_*.send_queue_length sv_*.send_urg

```
--Add the data, urgent and push information provided by the ULP
--in a SEND to the send_queue of the state vector.

 Enqueue contents of from_ULP.data to sv_*.send_queue, stamping each
    data octet with the current time;
 sv_*.send_queue_length := sv_*.send_queue_length + from_ULP.data_length;

 if (from_ULP.push_flag = TRUE)
 then sv_*.send_push := sv_*.send_queue_length;

 if (from_ULP.urgent_flag = TRUE)
 then sv_*.send_urg sv_*.send_queue_length;
```

3.2.6.3.2 assign new lcn The assign_new_lcn action procedure assigns a local connection name not currently used for a new open request and subsequent connection.
The data effects of this procedure are:

- Data examined: internal resources

- Data modified: none

```
--The procedure returns the value to be used as the new
--local connection name.
```

3.2.6.3.3 error(local connection name, error description) The error action procedure copies the local connection name and error description text supplied by parameter into the to_ULP structure. The service response field is assigned to ERROR for subsequent transfer to the ULP.
The data effects of the procedure are:

- Data examined: procedure parameters

- Date modified: to_ULP.lcn to_ULP.error_desc

```
to_ULP.lcn := local_connection_name;
to_ULP.error_desc := error_description;
to_ULP.service_response := ERROR;
```

3.2.6.3.4 initialize(sv_*); The initialize action procedure clears all values of the state vector named by parameter.
The data effects of this procedure are:

- Data examined: procedure parameter

- Data modified: all fields of sv_*

```
dequeue(sv_*.send_queue,sv_*.send_queue_length);
dequeue(sv_*.recv_queue,sv_*.recv_queue_length);
sv_* := null_state_vector;
```

3.2.6.3.5 open fail(local connection name) The open_fail action procedure copies the local connection name supplied by parameter, and the OPEN_FAIL service response into the to_ULP structure for subsequent transfer to the ULP.
The data effects of this procedure are:

- Data examined: procedure parameter

- Data modified: to_ULP.lcn to_ULP.service_response

```
to_ULP.lcn := local_connection_name;
to_ULP.service_response := OPEN_FAIL;
```

3.2.6.3.6 open_id(local_connection_name, source_port, source_address, destination_port, destination_addr) The open_id action procedure copies the parameters and the OPEN_ID service response into the to_ULP structure for subsequent transfer to the ULP.
The data effects of this procedure are:

- Data examined: procedure parameters

- Data modified:

to_ULP.lcn	to_ULP.service_response
to_ULP.source_port	to_ULP.destination_port
to_ULP.source_addr	to_ULP.destination_addr

```
to_ULP.lcn := local_connection_name;
to_ULP.source_port := source_port;
to_ULP.source_addr := source_address;
to_ULP.destination_port := destination_port;
to_ULP.destination_addr := destination_addr;
to_ULP.service_response := OPEN_ID;
```

3.2.6.3.7 open_success(local_connection_name) The open_success action procedure copies the local connection name supplied by parameter, and the OPEN_SUCCESS service response into the to_ULP structure for subsequent transfer to the ULP.
The data effects of this procedure are:

- Data examined: procedure parameter

- Data modified: to_ULP.lcn to_ULP.service_response

```
to_ULP.lcn := local_connection_name;
to_ULP.service_response := OPEN_SUCCESS;
```

3.2.6.3.8 terminate(local_connection_name, description) The terminate action procedure copies the local connection name and description supplied by parameter, and the TERMINATE service response into the to_ULP structure for subsequent transfer to the ULP.
The data effects of this procedure are:

- Data examined: procedure parameters

- Data modified:

to_ULP.service_response	to_ULP.lcn
to_ULP.error_description	

```
to_ULP.lcn := local_connection_name;
TO_ULP.error_desc := description;
to_ULP.service_response := TERMINATE;
```

3.2.6.3.9 record open parameters(ULP identifier, open mode) The record_open_parameters action procedure copies the values provided by the ULP in an open request to the state vector.
The data effects of this procedure are:

- Data examined:

from_ULP.source_port	from_ULP.precedence
from_ULP.source_addr	from_ULP.security
from_ULP.timeout	

- Data modified:

sv_*.source_port	sv_*.original_prec
sv_*.source_addr	sv_*.security
sv_*.destination_port	sv_*.timeout
sv_*.destination_addr	sv_*.open_mode

```
--Record the socket-pair and connection information
--provided in the open service request in the state_vector.

  sv_*.port := from_ULP.source_port;
  sv_*.addr := the address of this TCP;
  sv_*.open_mode := open_mode;

--Record timeout, security, and precedence for ULP *
--if provided, otherwise assign default values;

  if (from_ULP.security !!= NULL)
  then sv_*.security := from_ULP.security
  else sv_*.security := DEFAULT_SECURITY;

  if (from_ULP.precedence !!= NULL)
  then sv_*.original_prec := from_ULP.precedence
  else sv_*.original_prec := DEFAULT_PRECEDENCE;

  if (from_ULP.timeout !!= NULL)
  then sv_*.ulp_timeout := from_ULP.timeout
  else sv_*.ulp_timeout := DEFAULT_TIMEOUT;

  if (sv_*.open_mode !!= UNPASSIVE)
  then sv_*.destination_port := from_ULP.destination_port;
       sv_*.destination_addr := from_ULP.destination_addr;
  else sv_*.destination_port := NULL;
       sv_*.destination_addr := NULL;
```

3.2.6.3.10 try to deliver The try_to_deliver action procedure determines from the receive allocation, the receive queue size, and the receive push and urgent variables how much data to deliver to the local ULP. This procedure is called from several places for both ULP A and ULP B in the service machine definition. Where the sv_* notation is used, the appropriate state vector name should be replaced.
The data effects of this procedure are:

- Data examined: sv_*.source_port

- Data modified:

to_ULP.data	sv_*.recv_push
to_ULP.data_length	sv_*.recv_urg
to_ULP.urgent_flag	sv_*.recv_alloc
to_ULP.lcn	sv_*.recv_queue_length
sv_*.recv_queue	

```
--The amount of data delivered is based on the amount of pushed
--data waiting and the receive allocation.

  if (sv_*.recv_push !!= 0)
  then --As much pushed data allowed by the recv allocation
       --is delivered.

       if (sv_*.recv_alloc > sv_*.recv_push)
       then
            to_ULP.data_length := sv_*.recv_push;
            sv_*.recv_push := 0;
       else
            to_ULP.data_length := sv_*.recv_alloc;
            sv_*.recv_push := sv_*.recv_push - to_ULP.data_length;

  else  --Without a PUSH, there is no guarantee of delivery.
        --Deliver some amount of data less than or equal to receive
        --allocation (possibly none).

        to_ULP.data_length := some value;

  if (to_ULP.data_length !!= 0)
  then --Update state vector elements and prepare data and
       --and parameters for delivery.
       to_ULP.lcn := sv_*.lcn;

       --Urgent data cannot be delivered followed by non-urgent data.
       --If "end-of-urgent" falls in data to be delivered, make
       --two separate deliveries.
       if ((sv_*.recv_urg !!= 0) and
           (sv_*.recv_urg < to_ULP.data_length))
       then begin
            --Deliver urgent data alone.
            save := to_ULP.data_length;
            to_ULP.data_length := sv_*.recv_urg;
```

```
            sv_*.recv_urg := 0;
            to_ULP.urgent_flag := false;
            Dequeue to_ULP.data_length octets from sv_*.recv_queue
               and place in to_ULP.data;
            sv_*.recv_alloc := sv_*.recv_alloc - to_ULP.data_length;
            sv_*.recv_queue_length := sv_*.recv_queue_length -
                                              to_ULP.data_length;
            TRANSFER to_ULP to the ULP named by sv_*.source_port;

            --Prepare to deliver remaining non-urgent data.
            to_ULP.data_length := save;
            end;

      --If urgent data follows the data being delivered, inform ULP.
      if (sv_*.recv_urg > to_ULP.data_length)
      then
            to_ULP.urgent_flag := TRUE;
            sv_*.recv_urg := sv_*.recv_urg - to_ULP.data_length;
      else
            to_ULP.urgent_flag := FALSE;
            sv_*.recv_urg := 0;

      Dequeue to_ULP.data_length octets from sv_*.recv_queue
         and place in to_ULP.data;
      sv_*.recv_alloc := sv_*.recv_alloc - to_ULP.data_length;
      sv_*.recv_queue_length := sv_*.recv_queue_length -
                                        to_ULP.data_length;
      TRANSFER to_ULP to the ULP named by sv_*.source_port;
```

4. SERVICES REQUIRED FROM LOWER LAYER

This section describes the minimum set of services required of the network layer protocol by TCP.
The services required are:

- o data transfer service
- o generalized network service
- o error reporting service

A description of each service follows.

4.1 Data Transfer Service

The lower layer protocol must provide data transfer between TCP modules in a communication system. Such a system may consist of a single network or a set of interconnected networks forming an internet. Data must arrive at a destination with non-zero probability; some data loss may occur. The data transfer service is not required to preserve the order in which portions of data are supplied by the source upon delivery at the destination. Data delivered is not necessarily error-free.

The lower layer protocol must provide data transfer throughout the system. TCP need only supply global addressing and control information with each portion of data to be delivered. Routing and network specific characteristics are handled by the network layer protocol. For example, TCP need not be aware of current topology or packet size restrictions to transmit segments through a particular network.

4.2 Generalized Network Service

The lower layer protocol must provide a means for TCP to select from the transmission service qualities provided by the communication system for each portion of data delivered. The transmission quality parameters must include precedence. Also, the lower layer protocol must provide a means of labelling each portion of data with security information including security level, compartmentation, handling restrictions, and transmission control code (i.e., closed user groups).

4.3 Error Reporting Service

The lower layer protocol must provide error reports to TCP indicating discontinuation of the above services caused by catastrophic conditions in this or lower layer protocols.

5. LOWER LAYER SERVICE/INTERFACE SPECIFICATIONS

This section specifies the minimal subnetwork protocol services required by TCP and the interface through which those services are accessed. The first part defines the interaction primitives and their parameters for the lower interface. The second part contains the abstract machine specification of the lower layer services and interaction discipline.

5.1 INTERACTION PRIMITIVES

An interaction primitive defines the purpose and content of information exchanged between two protocol layers. Two kinds of primitives, based on the direction of information flow, are defined. Service requests pass information downward; service responses pass information upward. These primitives need not occur in pairs, nor in a synchronous manner. That is, a request does not necessarily elicit a "response"; a "response" may occur independently of a request.

The information associated with an interaction primitive falls into two categories: parameters and data. The parameters describe the data and indicate how the data is to be treated. The data is not examined or modified. The format of interaction primitive information is implementation dependent and so is not specified.

A given TCP implementation may have slightly different interfaces imposed by the nature of the network or execution environment. Under such circumstances, the primitives can be modified to include more parameters or additional primitives can be defined. However, all TCPs must provide at least the interface specified below to guarantee that all TCP implementations can support the same protocol hierarchy.

5.1.1 Service Request Primitives

A single service request primitive is required from the network protocol, NET_SEND.

5.1.1.1 NET SEND The NET_SEND primitive contains complete control information for each unit of data to be delivered. TCP passes in a NET_SEND at least the following information:

- o source address - address of TCP sending data
- o destination address - address of the TCP to receive data
- o protocol - identifier assigned to recipient TCP
- o type of service indicators - relative transmission quality associated with unit of data
  - precedence - one of eight levels : (P0, P1, P2, P3, P4, P5, P6, P7) where P0 <= P1 <= P2 <= P3 <= P4 <= P5 <= P6 <= P7

- reliability - one of two levels : (R0, R1) where R0 = normal reliability and R1 = high reliability.

- delay - one of two levels : (D0, D1) where D0 = normal delay and D1 = low delay.

- throughput - one of two levels : (T0, T1) where T0 = normal throughput and T1 = high throughput.

o identifier - value distinguishing this portion of data from others sent by this ULP.

o don't fragment indicator - flag showing whether the network protocol can fragment data to accomplish delivery

o time to live - value in seconds indicating maximum lifetime of data within the network

o data length - length of data being transmitted

o option data - options requested by TCP from those supported by network protocol including at least security labelling. (The Internet Protocol supports security labelling, source routing, return routing, stream identification, and time stamping[8].)

o data - present when data length is greater than zero.

5.1.2 Service Response Primitives
A single service response primitive, DELIVER, is required of the network delivery service.

5.1.2.1 NET_DELIVER The NET_DELIVER primitive contains the data passed by a source TCP in a NET_SEND, along with addressing, quality of service, and option information. TCP receives in a NET_DELIVER at least the following information:

o source address - address of sending TCP

o destination address - address of the recipient TCP

o protocol - identifier assigned to TCP as supplied by the sending TCP

o type of service indicators - relative transmission quality associated with unit of data

- precedence - one of eight levels : (P0, P1, P2, P3, P4, P5, P6, P7) where P0 <= P1 <= P2 <= P3 <= P4 <= P5 <= P6 <= P7

- reliability - one of two levels : (R0, R1) where R0 = normal reliability and R1 = high reliability.

- delay - one of two levels : (D0, D1) where D0 = normal delay and D1 = low delay.

- throughput - one of two levels : (T0, T1) where T0 = normal throughput and T1 = high throughput.

o data length - length of received data (possibly zero)

o option data - options requested by source TCP as supported by the network including at least security labelling. (The Internet Protocol supports security labelling, source routing, return routing, stream identification, and time stamping options[8].)

o data - present when data length is greater than zero.

In addition, a NET_DELIVER may contain error reports from the network protocol either together with parameters and data listed above, or, independently of that information. The details of the error reports are network dependent.

5.2 EXTENDED STATE MACHINE SPECIFICATION OF SERVICES REQUIRED FROM LOWER LAYER

The extended state machine in this section defines the behavior of the entire network protocol service machine from the perspective of TCP. This service machine is modelled as a "black box" whose internal actions are hidden from the TCPs using the network protocol's services. The TCP-pair provides stimuli, in the form of service requests, and receives the resulting network protocol reactions, in the form of service responses.

An abstract machine definition is composed of a machine identifier, a state diagram, a state vector, a set of data structures, an event list, and an events and actions correspondence.

5.2.1 Machine Instantiation Identifier

Each upper interface state machine is uniquely identified by the four interaction primitive parameters:

o source address

o destination address

o protocol

o identifier

One state machine instance exists for the NET_SEND and NET_DELIVER primitives whose parameters carry the same values.

5.2.2 State Diagram
The upper interface state machine has a single state which never changes. No diagram is needed.

5.2.3 State Vector
The upper interface state machine has a single state which never changes. No state vector is needed.

5.2.4 Data Structures
For clarity in the events and actions section, data structures are declared for the interaction primitives and their parameters. A subset of ADA data constructs, common to most high level languages, is used. However, a data structure may be partially typed or completely untyped where specific formats or data types are implementation dependent.

5.2.4.1 to NET The to_NET structure holds the interface parameters and data associated with the NET_SEND primitive specified above. This structure directly corresponds to the to_NET structure declared in Section 6.3.4.4 of the mechanism definition. The to_NET structure is declared as:

```
type to_NET_type is
  record
    source_addr
    destination_addr
    protocol
    type_of_service is
      record
          precedence
          reliability
          delay
          throughput
      end record;
    identifier
    time_to_live
    dont_fragment
    length
    data
    options
  end record;
```

5.2.4.2 from NET The from_NET structure holds interface parameters and data associated with the NET_DELIVER primitive, as specified in Section 5.1.2 above. This structure directly corresponds to the from_NET structure declared in Section 6.3.4.5 of the mechanism definition. The from_NET structure is declared as:

```
type from_NET_type is
  record
    source_addr
```

```
        destination_addr
        protocol
        type_of_service is
          record
              precedence
              reliability
              delay
              throughput
          end record;
        length
        data
        options
        error
      end record;
```

5.2.5 Event List

The events are drawn from the interaction primitives specified in Section 5.1 above. An event is composed of a service request primitive and an abstract timestamp to indicate the time of event initiation. The event list:

1. NET_SEND(to_NET) at time t

2. NULL - Although no service request is issued by TCP, certain conditions within network layer or lower layers produce a service response. These conditions can include duplication of data and subnet errors.

5.2.6 Events and Actions

The following section defines the set of possible actions elicited by each event.

EVENT = NET_SEND(to_NET) at time t

Actions:

1. NET_DELIVER from_NET at time t+N to TCP
   at destination to_NET.destination_addr with all of
   the following properties:

   a. The time elapsed during data transmission satisfies
      the time-to-live limit, i.e. N <= to_NET.time_to_live.

   b. The quality of data transmission is at least
      equal to the relative levels specified by
      to_NET.type_of_service.

   c. if (to_NET.dont_fragment = TRUE)
      then network layer fragmentation has not occurred
      in transit.

d. if (to_NET.options includes loose source routing)
then from_NET.data has visited in transit at least
the gateways named by the source provided by NET_SEND.

e. if (to_NET.options includes strict source routing)
then from_NET.data has visited in transit only
the gateways named by source route provided by NET_SEND.

f. if (to_NET.options includes record routing)
then the list of nodes visited in transit is
delivered in from_NET.

g. if (to_NET.options includes security labelling)
then the security label is delivered in from_NET.

h. if (to_NET.options includes stream identifier)
then the stream identifier is delivered in from_NET.

i. if (to_NET.options includes internet timestamp)
then the internet timestamp is delivered in from_NET.

OR,
2. NET_DELIVER to TCP source to_NET.source_addr indicating
one of the following error conditions:

a. destination to_NET.destination_addr unreachable

b. protocol to_NET.protocol unreachable

c. if (to_NET.dont_fragment = TRUE)
then fragmentation needed but prohibited

d. if (to_NET.options contains any option)
then parameter problem with option.

OR,
3. no action

EVENT = NULL

Actions:

1. NET_DELIVER to TCP at source to_NET.source_addr
indicating the following error condition:

a. error conditions in network or lower layers

OR,

2. NET_DELIVER from_NET at time t+N to TCP at destination

to_NET.destination_addr with all of the following properties:

a. The time elapsed during data transmission satisfies the time-to-live limit, i.e. N <= from_NET.time_to_live.

b. The quality of data transmission is at least equal to the relative levels specified by from_NET.type_of_service.

c. if (from_NET.dont_fragment = TRUE) then network layer fragmentation has not occurred in transit.

d. if (from_NET.options includes loose source routing) then to_NET.data has visited in transit at least the gateways named by the source provided by NET_SEND.

e. if (from_NET.options includes strict source routing) then to_NET.data has visited in transit only the gateways named by source route provided by NET_SEND.

f. if (from_NET.options includes record routing) then the list of nodes visited in transit is delivered in to_NET.

g. if (from_NET.options includes security labelling) then the security label is delivered in to_NET.

h. if (from_NET.options includes stream identifier) then the stream identifier is delivered in to_NET.

i. if (from_NET.options includes internet timestamp) then the internet timestamp is delivered in to_NET.

6. TCP ENTITY SPECIFICATION

This section defines the mechanisms of each TCP entity supporting the services provided by the TCP service machine. The first subsection motivates the specific mechanisms chosen and discusses the underlying philosophy of those choices. The second subsection defines the format and use of the TCP segment header fields. The last subsection specifies an extended state machine representation of the TCP entity.

6.1 OVERVIEW OF TCP MECHANISMS

The TCP mechanisms are motivated by TCP services, described in Section 2:

- o multiplexing service
- o connection management services
- o data transport service
- o error reporting service

Each service could be supported by any of several mechanisms. The selection of mechanisms is guided by design standards including simplicity, generality, flexibility, and efficiency. The mechanism descriptions identify the service or services supported and explain how the mechanisms work.

This overview begins with an introduction to some basic terminology used throughout the TCP entity mechanisms discussions. The mechanisms present in a TCP entity are:

flow control windows
duplicate and out-of-order data detection
positive acknowledgements with retransmission
checksum
push
urgent
ULP timeout
security and precedence
multi-addressing
passive and active open requests
three-way handshake for SYN exchange
open request matching
three-way handshake for FIN exchange
resets

6.1.1 Background and Terminology

This section presents the terminology used in the mechanism descriptions. The concept of sequence numbers and sequence space, the variables maintained in a state vector (defined in Section 6.3.4.1) and segment header fields (defined

in Section 6.2) are introduced. Also presented is a list of the states within the TCP state machine (defined in Section 6.3).

6.1.1.1 Sequence Numbers A fundamental notion in the design of the TCP entity is that every octet of data sent over a connection has a sequence number. These sequence numbers are used by several mechanisms (data ordering, duplicate detection, positive acknowledgement with retransmission, and flow control windows) to provide reliable, ordered data transfer.

The sequence number carried in a TCP header is a four octet value designating the sequence number of the first octet of data in the segment. Each successive data octet is numbered sequentially. Thus, each segment is bound to as many consecutive sequence numbers as there are octets of data in the segment.

The numbering scheme is extended to include certain control information as well. This is achieved by implicitly including some control flags in the sequence space so they can reliably transmitted without confusion (i.e., one and only one copy of the control will be acted upon). Control information is not physically carried in the segment data space. Consequently, we must adopt rules for implicitly assigning sequence numbers to control. The "SYN" and "FIN" flags are the only controls requiring this protection. These controls are used only at connection opening and closing. For sequence number purposes, the "SYN" is considered to occur before the first actual data octet of the segment in which it occurs. When a "SYN" is present then SEG.SEQ is the sequence number of the "SYN." The FIN is considered to occur after the last actual data octet in a segment in which it occurs. The segment length (LENGTH) includes both data and sequence space occupying controls.

It is essential to remember that the actual sequence number space, ranging from 0 to 2**32-1, is finite though very large. Because the space is finite, all arithmetic dealing with sequence numbers must be performed modulo 2**32. This unsigned arithmetic preserves the relationship of sequence numbers as they wrap around from 2**32-1 to 0 again.

6.1.1.2 Connection Sequence Variables To maintain a connection, a TCP entity records and updates connection status information in a state vector. (This is also called a Transmission Control Block, or TCB.) Among the status information stored in the state vector are sequence variables describing the data exchange over the connection. A connection carries data in two directions, and so each TCP entity maintains sequence variables for both the data it sends and the data it receives.

Send Variables Send variables are used to track the status of the send data stream with regard to acknowledgements, urgent data, pushed data, window size and position, and the initial sequence number. This list is a subset of the complete list of all send variables appearing in the state vector definition, Section 6.3.3.

1. SEND_NEXT - send next, the sequence number of the next octet of data to be sent

2. SEND_UNA - send unacknowledged, all octets up to but not including this sequence number have been acknowledged.

3. SEND_WNDW - send window, the number of data octets currently allowed to be sent relative to SEND_UNA.

4. SEND_URG - send urgent point, the sequence number of the last octet of urgent data.

5. SEND_PUSH - send push point, the sequence number of the last octet of pushed data.

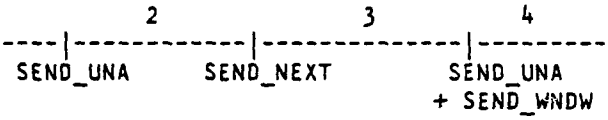
6. SEND_LASTUP1 - last window update one, the sequence number carried by the incoming segment used for last window update.

7. SEND_LASTUP2 - last window update two, the acknowledgement number carried by the incoming segment used for last window update.

8. SEND_ISN - initial send sequence number, the sequence number of the SYN sent on this connection.

These names correspond to the state vector elements defined in Section 6.3.3.

Send Sequence Space If the space of send sequence numbers is pictured as a number line, the following diagram shows the relationships among some of the variables defined above.



1 - old sequence numbers which have been acknowledged
2 - sequence numbers of sent but as yet unacknowledged data
3 - sequence numbers allowed for new data transmission
(i.e. the unused send window)
4 - future sequence numbers which are not yet allowed

Receive Variables The receive variables track the receive data stream in regard to acknowledgements, urgent data, pushed data, window size and position, and initial sequence number. These variables are a subset of the state vector elements defined in Section 6.3.3.

1. RECV_NEXT - receive next, the sequence number of the next data octet to be received.

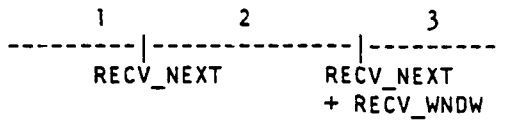
2. RECV_WNDW - the number of data octets that can currently be received starting from RECV_NEXT.

3. RECV_URG - receive urgent point, the sequence number of the last octet of urgent data.

4. RECV_PUSH - receive push point, the sequence number of the last octet of pushed data.

5. RECV_ISN - initial receive-sequence number, the sequence number of the SYN received from the remote TCP.

Receive Sequence Space If the space of receive sequence numbers is pictured as a number line, the following diagram shows the relationships among some of the variables defined above.



1 - old sequence numbers which have already been accepted
2 - sequence numbers allowed to be received
(i.e., the receive window)
3 - future sequence numbers not yet allowed

6.1.1.3 Current Segment Variables TCP entities communicate in units of exchange called segments. A segment is made up of a header, containing addressing and control information, and a text area, containing a portion of the send or receive data streams. A formal definition of the segment header format appears in Section 6.2. The following header fields and related values are used in the mechanism descriptions.

1. SEG.SEQ - segment sequence number, the sequence number carried in the segment header. It may number the first octet of carried data, number a sequence control flag, or (in an empty segment) indicate the next octet to be sent.

2. SEG.ACK - segment acknowledgement number, the acknowledgement from the sending TCP. That is, the next sequence number expected from the receiving TCP.

3. SEG.WNDW - segment window, the current number of octets that the sending TCP will accept as counted from SEG.ACK.

4. SEG.URGPTR - segment urgent pointer, the number of data octets remaining before the end of the urgent data, as counted from SEG.SEQ.

5. PRECEDENCE - segment precedence level, supplied as a service response parameter.

6. LENGTH - segment length, number of octets of header and text carried in the segment. This value is supplied as a service response parameter.

6.1.1.4 Connection States A TCP connection progresses through three phases: opening (or synchronization), maintenance, and closing. The three phases are broken down further into states which represent significant stages in the handshake mechanisms of connection opening and closing. These states correspond to the values assumed by the primary element of the state vector structure, sv.state. The TCP entity states are:

1. LISTEN - after a passive open request from the local ULP, represents waiting for a connection request from a remote TCP (in the form of a SYN segment).

2. SYN_SENT - after an active open request from the local ULP and having sent an open request (i.e., a SYN), represents waiting for a matching connection open request (i.e., another SYN) from the remote TCP.

3. SYN_RECEIVED - represents waiting for a confirming connection request acknowledgement (i.e., the ACK of the SYN) after having both received and sent connection requests.

4. ESTABLISHED - represents an open connection on which data can be passed between ULPs in both directions.

5. FIN_WAIT1 - after a close service request from the local ULP, represents waiting for either a close request (in the form of a FIN segment) from the remote TCP, or an acknowledgement of the close request already sent (i.e., an ACK of the FIN). Data received from remote TCP is delivered to the local ULP.

6. FIN_WAIT2 - represents waiting for a connection termination request (i.e., a FIN) from the remote TCP. Data received from remote TCP is delivered to the local ULP.

7. CLOSE_WAIT - represents having received a connection close request (i.e., a FIN) from the remote TCP and waiting for a connection close request from the local ULP. Data sent by the local ULP is sent to the remote TCP.

8. LAST_ACK - represents having both sent and received a connection close request, having acknowledged the remote close request, and waiting for the last acknowledgment from the remote TCP.

9. CLOSING - represents waiting for the acknowledgement of a connection close request (i.e., an ACK of the FIN) from the remote TCP.

10. TIME_WAIT - represents waiting for enough time to pass to ensure the remote TCP has received the acknowledgement of its connection close request.

11. CLOSED - represents no connection.

The full definition of the TCP states, events, and processing appears in Section 6.3.

6.1.2 Flow Control Window

TCP provides a flow control mechanism, called a window, to enable a receiving TCP entity to govern the amount of data transmitted by a sending TCP entity. A window is an "absolute" flow control technique. Absolute flow control defines an interval of sequence numbers corresponding to the amount of data an entity is willing to accept. This technique prevents ambiguity introduced by duplicate segments because permission to transmit is specified as a specific range of sequence numbers rather than an incremental value.

The receiving TCP maintains the amount of data allowed for acceptance in the receive variable RECV_WNDW. This value is bound to the receive sequence space beginning at RECV_NEXT, the next expected data octet. A TCP entity communicates its current receive window to the remote TCP by placing the window in each outbound segment header in the following manner. The window field of the segment header, SEG.WINDOW, is a positive integer value expressing the number of acceptable data octets. The acknowledgement number in the segment header, SEG.ACK, associates that quantity to the receive sequence space. Thus, the receive window starts with SEG.ACK and continues through the number of octets indicated by SEG.WINDOW.

As each incoming segment is validated by sequence number and acknowledgement (Sections 6.1.3 and 6.1.4), the TCP entity records the window size in the send variable, SEND_WNDW.

6.1.2.1 Shrinking Windows A TCP entity is strongly discouraged from "shrinking" its receive window. A window is said to shrink when a TCP entity advertises a large window and subsequently advertises a smaller one without having accepted the difference in data. Such behavior complicates the send data algorithms of the peer entity. For example, a sending TCP may act upon a large window allocation by sending all of the advertised amount. When the window shrinks, data already sent becomes outside the window. The sender must either set back the send variables and remove data from the retransmission queue to "un-send" the data, or else ignore the smaller window. The robustness principle mandates that although a TCP entity does not shrink its own receive window, it will be prepared for such behavior by other entities.

6.1.2.2 Zero Windows Windows can close, that is become zero in length, when a receiving TCP has no more room to receive data, either because the ULP has stopped accepting or because system resources have been temporarily exhausted. In this situation, the sending TCP normally would not send data. And, if no data is generated by the other ULP, the sending entity will receive no new window updates. Without special mechanisms, zero windows could halt data transfer.

With a zero send window, a sending TCP must be prepared to accept from the local and send to the remote TCP at least one octet of new data. Also, a sending TCP must transmit segments at regular intervals into the zero window in order to guarantee that the re-opening of the receive window will be reliably reported. The recommended transmission interval in this situation is two minutes.

With a zero receive window, a TCP entity receiving a segment with data must

still send an acknowledgement showing its next expected sequence number and current window even though it does not accept the data. If the receiving TCP emits an empty ACK segment when opening its receive window, it may resume receiving data more quickly.

Even with a zero receive window, a TCP must process the ACK, RST, and URG fields of all acceptable incoming segments.

6.1.2.3 Window Updates with One-way Data Flow In a connection with data flowing primarily in one direction, the window information will be carried in segments marked with the same sequence number. If such segments arrive out-of-order, they cannot be reordered. This situation is not serious, but it does allow the window information to occasionally be based on old reports from the receiver.

A strategy to avoid this problem is to check both the sequence number and the acknowledgement number when deciding to update the send window. That is, use the window information from segments carrying either a higher sequence number than previously seen, or the same sequence number and the highest acknowledgement number. The highest sequence number of an incoming segment used for a window update is recorded in the send variable SEND_LASTUP1; the highest acknowledgement number in SEND_LASTUP2.

6.1.2.4 Window Management Suggestions A TCP entity's method of managing its window has significant influence on performance. The following sections discuss certain window management policies and their effects.

Window Size vs. Actual Capacity In general, advertised window size is based on the amount of available receive storage. Although indicating large windows encourages transmissions, false window promises can degrade performance. If the window is larger than actual storage capacity, more data may arrive than can be accepted. The excess data is discarded, causing its retransmission, adding unnecessarily to the load on the communications system and the sending TCP.

Small Windows Allocating very small windows causes data to be transmitted in many small segments. Better performance may be achieved using fewer large segments. In general, if both sending and receiving window management algorithms actively attempt to combine small window allocations into larger windows, the tendency toward small segments can be avoided.

One suggestion to avoid small windows is for a receiving TCP to defer updating a window until an allocation is at least X percent of the maximum allocation possible for the connection (where X might be 20 to 40). Thus, the TCP could be send an ACK when a segment arrives (without updating the window information), and later send another ACK with the larger window. Another suggestion is for the sending TCP to avoid sending small segments by waiting until the window is "large" before sending data. (Note that acknowledgements should not be delayed or unnecessary retransmissions will result.)

6.1.3 Duplicate and Out-of-order Data Detection

The network protocol layer may duplicate or change the order of segments submitted by TCP for transmission. To compensate, a TCP entity uses sequence numbers to detect out-of-order and duplicate segments. Duplicate segments are discarded. Segments arriving out of order may, depending on implementation choices, be either discarded or saved for subsequent processing.

The duplicate detection and sequencing algorithms rely on the unique binding of segment data to sequence space. The algorithms are based on the assumption that all 2**32 sequence number values are not cycled through before the segment data bound to those sequence numbers has been delivered and acknowledged by the receiver and all duplicate copies of the segments have "drained" from the internet. Without such an assumption, two distinct TCP segments could conceivably be assigned the same or overlapping sequence numbers, causing confusion at the receiver as to which data is new and which is old. This topic is discussed in [10].

A sending TCP entity keeps track of the sequence number of the next data octet to send in the variable SEND_NEXT. In each outgoing segment, the entity records the sequence number of the first data octet in the segment header field, SEG.SEQ, and advances SEND_NEXT by the total amount of data carried in the segment. A receiving TCP entity keeps track of the sequence number of the next data octet expected in the variable RECV_NEXT. That value and the variable RECV_WNDW represent the receive window, or interval of acceptable data octets. This interval is compared against incoming segment sequence numbers to determine their "acceptability."

An incoming segment is defined to be acceptable if any data it carries falls within the receive window. If the segment does not carry data, the segment sequence number must fall within the receive window. When the receive window is zero, a segment is acceptable if its sequence number equal the next expected sequence number, RECV_NEXT.

The processing of unacceptable segments is discussed in 6.1.3.

The control information, including valid acknowledgment, window and urgent information, must be used from every acceptable segment. However, the policy for taking (i.e., adding to to the receive queue) the data of an acceptable segment can be approached in two ways. The first approach takes only in-order data. That is, only data octets with sequence numbers starting at RECV.NEXT and continuing through to either the end of the segment or the end of the receive window (whichever is shorter) are taken. The data octets of acceptable segments with sequence numbers starting beyond RECV.NEXT are not taken. This "in-order" approach allows immediate acknowledgment and delivery to the ULP.

The second approach, called in-window data acceptance, takes any data falling within the receive window. If the data is not contiguous with previously received data, it is saved for processing until the intervening data arrives. Thus, acknowledgment and delivery will be delayed until a contiguous interval of data arrives.

6.1.4 Positive Acknowledgement With Retransmission

Another mechanism compensating for network protocol behavior is positive acknowledgement with retransmission. This mechanism replaces data lost or damaged in transit through the use of sequence numbers and acknowledgements. The basic strategy with PAR is for a sending TCP to retransmit a segment at timed intervals until a positive acknowledgement is returned. The mechanism requirements for segment retransmission, acknowledgement acceptance, transmission intervals, and sequence variable manipulation are described below.

The PAR strategy requires TCP to keep copies of all segments in order on a "retransmission queue." As each segment is sent, a segment copy is placed on the end of the queue. The retransmission queue holds the data octets whose sequence numbers begin with SEND_UNA, the oldest unacknowledged sequence number, and ends with SEND_NEXT, the next octet to be sent. When all sent data has been acknowledged, SEND_UNA equals SEND_NEXT, and the retransmission queue is empty.

When data is placed on the retransmission queue, a timer is set for the interval expected to elapse before its acknowledgement returns. When a segment or an acknowledgement is lost, the retransmission timer will expire and the TCP will retransmit the unacknowledged data.

If the original segment was lost or discarded due to damage, the retransmitted segment is accepted as the original at the receiving TCP. If the acknowledgement was lost, the receiving TCP discards the retransmitted segment as a duplicate, but resends its acknowledgement.

6.1.4.1 Acknowledgement Generation Every TCP segment, excluding an initial SYN segment, must carry an acknowledgement indicating current receive variable information. Acknowledgements are carried in the TCP segment header in a four octet field designating the sequence number of the next expected data octet. The acknowledgement mechanism is cumulative so that an ACK of sequence number X indicates that all octets up to but not including X have been received. Thus, a TCP entity sets the ACK field of each outgoing segment to the value of RECV_NEXT, implicitly stating that it has successfully received every data octet up to that sequence number.

An acknowledgement does not guarantee that data has been delivered to the ULP, but only that the destination TCP has taken the responsibility to do so.

6.1.4.2 ACK Validation Incoming acknowledgments are compared with the send variables to determine their "acceptability." An "acceptable ACK" is one for which the inequality holds:

SEND_UNA =< SEG.ACK =< SEND_NEXT

In other words, the acknowledgement refers to data equal to or beyond that already acknowledged, and yet does not exceed the sequence number of data yet to be sent. If SEG.ACK < SEND_UNA, it is an old ACK and is unacceptable. If SEG.ACK > SEND_NEXT, it acknowledges data not yet sent, and so is

unacceptable.

When an acceptable ACK equals SEND_UNA, no new data is acknowledged but new window information may be present. When an acceptable ACK is greater than SEND_UNA, it becomes the new value for SEND_UNA.

6.1.4.3 Retransmission Queue Removals Acknowledgements are not only used to update SEND_WNDW and SEND_UNA, they are also processed with respect to the retransmission queue. When an ACK arrives fully acknowledging a segment on the retransmission queue, the segment copy is removed from the queue. An ACK is said to fully acknowledge a segment copy on the retransmission queue if the sum of the segment copy's sequence number and length is less than or equal to the acknowledgement number of the incoming segment.

6.1.4.4 Retransmission Strategies A TCP implementation may employ one of several retransmission strategies.

a. First-only retransmission - The TCP entity maintains one retransmission timer for the entire queue. When the retransmission timer expires, it sends the segment (or a segment's worth of data) at the front of the retransmission queue and resets the timer.

b. Batch retransmission - The TCP entity maintains one retransmission timer for the entire queue. When the retransmission timer expires, it sends all the segments on the retransmission queue and resets the timer.

c. Individual retransmission - The TCP entity maintains one timer for each segment on the retransmission queue. As the timers expire, the segments are retransmitted individually and their timers reset.

A brief discussion of retransmission strategy trade-offs and their relationship to the acceptance policy appears in Appendix A.

6.1.4.5 Retransmission Timeouts The value of the retransmission timer can have a large effect on the performance of both the connection and the network. A timeout interval that is too short results in unnecessary retransmissions, wasting both TCP processing time and network resources, while one that is too long results in poor throughput and poor response time for the ULP.

Ideally, the retransmission interval should equal exactly the time required for a segment to traverse the network to its destination, be processed, and its ACK to traverse the network back to the source. This sum is called the Round Trip Time (RTT). Realistically, however, this value is rarely known or constant. Instead, an approximation of this sum can be dynamically computed during the lifetime of a connection.

6.1.5 Checksum

The checksum mechanism supports error-free data transfer service by enabling detection of segments damaged in transit. A checksum value is computed for each outbound segment and placed in the header's checksum field. Similarly, the checksum of each incoming segment is computed and compared against the value of the header's checksum field. If the values do not match, the incoming segment is discarded without being acknowledged. Hence, a damaged segment appears the same as a lost segment and is compensated for by the PAR mechanism.

TCP uses a simple one's complement algorithm which covers the segment header, the segment data, and a "pseudo header." The pseudo header is made up of the source address, the destination address, TCP's protocol identifier [5], and the length of the TCP segment (excluding the pseudo header). By including the extra pseudo header information in the checksum, TCP protects itself from misdelivery by the network protocol.

The checksum algorithm is the 16 bit one's complement of the one's complement sum of all 16 bit words in the pseudo header, segment header, and the segment text. If a segment contains an odd number of octets, the last octet is padded on the right with zeros to form a 16-bit word for checksum purposes. While computing the checksum, the checksum field itself is replaced with zeros.

6.1.6 Push

The data that flows on a connection is conceptually a stream of octets. A sending TCP is allowed to collect data from the sending ULP and to segment and send the data at its own convenience. The sending ULP has no way of knowing if the data has been sent or is retained by the local TCP or remote TCP while waiting for a more suitable segment or delivery size. This mechanism enables a ULP to push data through both the local and remote TCP entities.

When "push" flag is set in a SEND request, the sending TCP segments and sends all internally stored data within flow control limits. Upon receipt of a pushed segment, the receiving TCP must promptly deliver the pushed data to the receiving ULP.

Successive pushes may not be preserved because two or more units of pushed data may be joined into a single pushed unit by either the sending or receiving TCP. Pushes are not visible to the receiving ULP and are not intended to serve as a record boundary marker.

6.1.7 Urgent

TCP provides a means to communicate to a receiving ULP that some point in the upcoming data stream has been marked urgent by the sending ULP. Also, the receiving TCP can indicate when all the currently know urgent data has been delivered to the receiving ULP. The objective of the TCP urgent mechanism is to enable the sending ULP to stimulate the receiving ULP to accept some urgent data. TCP does not define what the ULP is required to do with the urgent state information, but the general notion is that the receiving ULP will take action to process the intervening data quickly.

The urgent mechanism permits a point in the data stream to be designated as the end of urgent information. Whenever this point is in advance of the variable RECV_NEXT at the receiving TCP, that TCP must tell the ULP to go into "urgent mode;" when the receive sequence number catches up to the urgent pointer, the TCP must tell the ULP to go into "normal mode." If the point is updated while the ULP is in urgent mode, the update will be invisible to the ULP. Note that urgent data cannot be delivered together with any non-urgent data that may follow.

The mechanism employs an urgent field which is carried in all segments transmitted. The URG control flag indicates that the urgent field is meaningful. The urgent field must be added to the segment sequence number to yield the sequence number of the last octet of urgent data. The absence of this flag indicates that there is no urgent data outstanding.

To send an urgent indication the ULP must also send at least one data octet. If the sending ULP also indicates a push, timely delivery of the urgent information to the destination process is enhanced. When an urgent indication appears in a Send service request but the send window does not allow data to be sent immediately, the TCP should send an empty ACK segment with the new urgent information.

6.1.8 ULP Timeout

The timeout allows a ULP to set up a timeout for all data submitted to the TCP entity. If some data is not successfully delivered to the destination within the timeout period, the TCP entity will terminate the connection.

The timeout appears as an optional parameter in the open requests and the send request. Upon receiving either an active open request, or a SYN segment after a passive request, the TCP entity must maintain a timer set for the interval specified by the ULP. As acknowledgements arrive from the remote TCP, the timer is cancelled and set again for the timeout interval. As a parameter of the SEND request, the timeout can change during connection lifetime. If the timeout is reduced below the age of data waiting to be acknowledged, the connection is terminated and the ULP is informed.

6.1.9 Security and Precedence

TCP makes use of the Internet Protocol (IP) [8] options to provide security and precedence on a per connection basis. The security and precedence parameters used in TCP are those defined in IP. Throughout this TCP specification the term "security information" indicates the security parameters used in IP, including security level, compartment, user group, and handling restrictions.

In order for a TCP connection to be established, the modules at each end of the connection must agree on the security information and precedence to be associated with the connection. When system requirements allow a choice of precedence levels and security information, such as in a multilevel secure environment, these parameters may be negotiated; for other systems that operate at a single security level or in an unclassified environment, predetermined values must be used.

The precedence level of the connection is negotiated through the exchange of lower bounds by each end during connection opening. The higher of the two values is assigned to the connection. If it is impossible for the end with the lower precedence to raise its level to the higher, or to get the security information to match exactly, the connection must be rejected. The inability to match security information or precedence levels is indicated by the receipt of segments after the connection opening with the non-matching information. The connection is then rejected by sending a reset. In addition to sending a reset, the connection attempt with mismatched security information may be reported or recorded in accordance with local standard operating procedures.

After the connection is established, the TCP modules must mark outgoing segments with the agreed security information and precedence level. Any incoming segment with security information or precedence level not exactly matching that of the connection causes the termination of the connection. A reset is sent to the remote TCP and the local ULP is informed of the error.

6.1.10 Multiplexing

TCP provides a set of addresses, called port identifiers, to allow for many ULPs within a single host to use TCP communication facilities simultaneously and to identify the separate data streams that a ULP may request. Port identifiers are selected independently by each TCP entity. To provide unique addresses, TCP concatenates an internet address identifying its internet location to a port identifier creating a "socket." Thus, sockets are unique throughout the internetwork and a pair of sockets can uniquely identify each TCP connection. A socket may participate in many connections to different foreign sockets.

TCPs are free to associate ports with processes however they choose. However, several basic concepts are necessary in any implementation. There are "well-known" sockets which a TCP entity associates only with the "appropriate" ULP by some means. Well-known sockets are a convenient mechanism for a priori associating a socket address with a standard service. For instance, the "Telnet-Server" process is permanently assigned to a particular socket, and other sockets are reserved for File Transfer, Remote Job Entry, Text Generator, Echoer, and Sink processes (the last three being for test purposes). A socket address might be reserved for access to a "Look-Up" service which would return the specific socket at which a newly created service would be provided. The concept of a well-known socket is part of the TCP specification, but the assignment of sockets to services is outside this specification [5].

6.1.11 Connection Opening Mechanisms

Several mechanisms are used to establish connections between two TCP entities. These mechanisms, including open requests, sequence number synchronization, and initial sequence number generation, are discussed below.

6.1.11.1 Connection Open Requests TCP provides a ULP with two ways of opening a connection, called passive open requests and active open requests. The open requests have certain parameters including the local socket and foreign socket naming the connection.

With a passive open request, the TCP entity assigns a state vector for the connection variables, returns a local connection name, and becomes "receptive" to connections with other ULPs. The foreign socket parameter in a passive open request may be either fully specified or unspecified. That is, when the foreign socket parameter is set to a specific socket value, only the ULP with that socket identifier can be connected. If the foreign socket is unspecified (denoted by all zeros) any ULP can be connected. Such unspecified foreign sockets are allowed only on passive open requests. A service ULP that wished to provide services for unknown other ULPs would issue an unspecifed passive open request, supplying its own well-known socket for the local socket.

With an active open request, the TCP entity not only assigns a state vector and a local connection name, but also actively initiates the connection by sending a SYN segment. A connection is initiated by the rendezvous of an arriving segment containing a SYN and a waiting state vector.

The matching of local and foreign sockets determines when a connection has been initiated. There are two principal cases for matching the sockets in the local open requests to the foreign sockets in arriving SYN segments. In the first case, the local open has fully specified the foreign socket so the match must be exact. In the second case, the local passive open has left the foreign socket unspecified so any foreign socket is acceptable as long as the local sockets match. Other possibilities, left up to the implementor, include partially restricted matches.

If there are several pending open requests with the same local socket, a foreign active open will be matched to to a fully specified open, if one exists, before selecting an unspecified passive open.

6.1.11.2 Three-Way Handshake The "three-way handshake" is the mechanism used to establish a connection. This procedure normally is initiated by one TCP and responded to by another TCP. The procedure also works if two TCPs simultaneously initiate the procedure. When two ULPs wish to to communicate, they issue open requests as described above, instructing their TCPs to initialize and synchronize the mechanism information on each side. However, the potentially unreliable network layer can complicate the process of synchronization. Delayed or duplicate segments from previous connection attempts might be mistaken for new ones. A handshake procedure with clock based sequence numbers is used in connection opening to reduce the possibility of such false connections.

In the simplest handshake between an active open request and a passive open request, the TCP pair synchronizes sequence numbers by exchanging three segments. The actively opened TCP entity emits a segment marked with a synchronize control flag, called a "SYN" segment, which is matched at the receiving TCP entity to the passive open request. The receiving TCP entity emits its own SYN also carrying an acknowledgement of the first SYN. That segment is responded to with an acknowledgement. Thus, a three segment exchange establishes the connection.

When simultaneous active open requests initiate the connection each TCP

receives a SYN segment which carries no acknowledgement after it has sent a SYN. Each respond with an acknowledging segment and a connection is established in four exchanges. Of course, the arrival of an old duplicate SYN segment can potentially make it appear, to the recipient, that a simultaneous connection initiation is in progress. Proper use of "reset" segments will disambiguate these cases.

Several examples of connection initiation follow. Although these examples do not show connection synchronization using data-carrying segments, this is perfectly legitimate, so long as the receiving TCP doesn't deliver the data to the ULP until it is clear the data is valid (i.e., the data must be buffered at the receiver until the connection reaches the ESTABLISHED state). The three-way handshake reduces the possibility of false connections. It is the implementation of a trade-off between memory and messages to provide information for this checking.

The simplest three-way handshake is shown in the scenario, Section 1.1. Other examples are shown below. The figures should be interpreted in the following way. Each line is numbered for reference purposes. Right arrows (-->) indicate departure of a TCP segment from TCP A to TCP B, or arrival of a segment at B from A. Left arrows (<--) indicate the reverse. Ellipsis (...) indicates a segment which is still in the network (delayed). An "XXX" indicates a segment which is lost or rejected. Comments appear in parentheses. TCP states represent the state AFTER the departure or arrival of the segment (whose contents are shown in the center of each line). Segment contents are shown in abbreviated form, with sequence number, control flags, and ACK field. Other fields such as window, addresses, lengths, and text have been left out in the interest of clarity.

Simultaneous Connection Initiation Simultaneous initiation is only slightly more complex than a three-way handshake. Each TCP cycles from CLOSED to SYN-SENT to SYN-RECEIVED to ESTABLISHED.

	TCP A			TCP B
1.	CLOSED			CLOSED
2.	SYN-SENT	-->	<SEQ=100><CTL=SYN>	...
3.	SYN-RECEIVED	<--	<SEQ=300><CTL=SYN>	<-- SYN-SENT
4.		...	<SEQ=100><CTL=SYN>	--> SYN-RECEIVED
5.	SYN-RECEIVED	-->	<SEQ=100><ACK=301><CTL=SYN,ACK>	...
6.	ESTABLISHED	<--	<SEQ=300><ACK=101><CTL=SYN,ACK>	<-- SYN-RECEIVED
7.		...	<SEQ=101><ACK=301><CTL=ACK>	--> ESTABLISHED

Simultaneous Connection Synchronization

The principal reason for the three-way handshake is to prevent old duplicate connection initiations from causing confusion. To deal with this, a special control message, reset, is used. If the receiving TCP is in a nonsynchronized state (i.e., SYN-SENT, SYN-RECEIVED), it returns to LISTEN on receiving an acceptable reset. If the TCP is in one of the synchronized states (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), it aborts the connection and informs its ULP. This case is discussed under "half-open" connections below.

Old Duplicate SYN Detection As a simple example of recovery from old duplicates, consider the following figure. At line 3, an old duplicate SYN arrives at TCP B. TCP B cannot tell that this is an old duplicate, so it responds normally (line 4). TCP A detects that the ACK field is incorrect and returns a RST (reset) with its SEQ field selected to make the segment believable. TCP B, on receiving the RST, returns to the LISTEN state. When the original SYN finally arrives at line 6, the synchronization proceeds normally. If the SYN at line 6 had arrived before the RST, a more complex exchange might have occurred with RSTs sent in both directions.

	TCP A		TCP B
1.	CLOSED		LISTEN
2.	SYN-SENT	--> <SEQ=100><CTL=SYN>	...
3.	(duplicate)	... <SEQ=90><CTL=SYN>	--> SYN-RECEIVED
4.	SYN-SENT	<-- <SEQ=300><ACK=91><CTL=SYN,ACK>	<-- SYN-RECEIVED
5.	SYN-SENT	--> <SEQ=91><CTL=RST>	--> LISTEN
6.		... <SEQ=100><CTL=SYN>	--> SYN-RECEIVED
7.	SYN-SENT	<-- <SEQ=400><ACK=101><CTL=SYN,ACK>	<-- SYN-RECEIVED
8.	ESTABLISHED	--> <SEQ=101><ACK=401><CTL=ACK>	--> ESTABLISHED

Recovery from Old Duplicate SYN

Half-Open Connections and Other Anomalies An established connection is said to be "half-open" if one of the TCPs has closed or aborted the connection at its end without the knowledge of the other, or if the two ends of the connection have become desynchronized owing to a crash that resulted in loss of memory. Such connections will automatically become reset if an attempt is made to send data in either direction. However, half-open connections are expected to be unusual, and the recovery procedure is somewhat involved.

If at site A the connection no longer exists, then an attempt by the ULP at site B to send any data on it will result in the site B TCP receiving a reset control message. Such a message indicates to the site B TCP that something is wrong, and it is expected to abort the connection.

Assume that two ULPs A and B are communicating with one another when a crash occurs causing loss of memory to A's TCP. Depending on the operating system supporting ULP A's TCP, it is likely that some error recovery mechanism exists. When the TCP is up again, ULP A is likely to start again from the beginning or from a recovery point. As a result, ULP A will probably try to OPEN the connection again or try to SEND on the connection it believes open. In the latter case, it receives the error message "connection not open" from the local (ULP A's) TCP. In an attempt to establish the connection, ULP A's TCP will send a segment containing SYN. This scenario leads to the example shown in figure 10. After TCP A crashes, the ULP attempts to open the connection again. TCP B, in the meantime, thinks the connection is open.

```
     TCP A                                                  TCP B

1.   (CRASH)                                        (send 300,receive 100)

2.   CLOSED                                                 ESTABLISHED

3.   SYN-SENT --> <SEQ=400><CTL=SYN>                  --> (??)

4.   (!!!!)      <-- <SEQ=300><ACK=100><CTL=ACK>      <-- ESTABLISHED

5.   SYN-SENT --> <SEQ=100><CTL=RST>                  --> (Abort!!!!)

6.   SYN-SENT                                               CLOSED

7.   SYN-SENT --> <SEQ=400><CTL=SYN>                  -->
```

Half-Open Connection Discovery

When the SYN arrives at line 3, TCP B, being in a synchronized state, sees the incoming segment outside the window and responds with an acknowledgement indicating what sequence it next expects to hear (ACK 100). TCP A sees that this segment does not acknowledge anything it sent and, being unsynchronized, sends a reset (RST) because it has detected a half-open connection. TCP B aborts at line 5. TCP A will continue to try to establish the connection; the problem is now reduced to the basic 3-way handshake.

An interesting alternative case occurs when TCP A crashes and TCP B tries to send data on what it thinks is a synchronized connection. This is illustrated in the next figure. In this case, the data arriving at TCP A from TCP B (line 2) is unacceptable because no such connection exists, so TCP A sends a RST. The RST is acceptable so TCP B processes it and aborts the connection.

	TCP A		TCP B
1.	(CRASH)		(send 300,receive 100)
2.	(??)	<-- <SEQ=300><ACK=100><DATA=10><CTL=ACK>	<-- ESTABLISHED
3.		--> <SEQ=100><CTL=RST>	--> (ABORT!!!!)

Active Side Causes Half-Open Connection Discovery

In the following figure, TCPs A and B with passive opens are waiting for SYNs. An old duplicate arriving at TCP B (line 2) stirs B into action. A SYN-ACK is returned (line 3) and causes TCP A to generate a RST (the ACK in line 3 is not acceptable). TCP B accepts the reset and returns to its passive *LISTEN* state.

	TCP A		TCP B
1.	LISTEN		LISTEN
2.		... <SEQ=Z><CTL=SYN> -->	SYN-RECEIVED
3.	(??)	<-- <SEQ=X><ACK=Z+1><CTL=SYN,ACK> <--	SYN-RECEIVED
4.		--> <SEQ=Z+1><CTL=RST> -->	(return to LISTEN!!)
5.	LISTEN		LISTEN

Old Duplicate SYN Initiates a Reset on two Passive Sockets

A variety of other cases is possible, all of which are accounted for by the reset generation and processing.

6.1.11.3 Initial Sequence Number Selection *TCP imposes no restrictions on a* particular connection being used over and over again. A connection is only named by a pair of sockets. New instances of a connection will be referred to as *incarnations* of the connection. The problem that *arises* is how to *identify* duplicate segments from previous incarnations of the connection. This problem becomes apparent if the connection is being opened and closed in quick succession, or if the connection breaks with loss of memory and is then reestablished.

To avoid confusion, segments from one incarnation of a connection must not be used while the same sequence numbers may still be present in the network from an earlier incarnation. This must be assured, even if a TCP crashes and loses all knowledge of the sequence numbers it has been using. Thus, a clock-based initial sequence number generation procedure has been defined.

6.1.11.4 ISN Generator When new connections are created, an initial sequence number (ISN) generator is employed which selects a new 32-bit ISN. The generator is bound to a (possibly fictitious) 32-bit clock whose low order bit is incremented roughly every 4 microseconds. Thus, the ISN cycles approximately every 4.55 hours. Assuming segments will stay in the network no more than the Maximum Segment Lifetime (MSL) and that the MSL is less than 4.55 hours, ISNs will be unique. The advantages of this technique are discussed in [10].

6.1.12 Connection Closing Synchronization
Connection closing is handled similarly to connection establishment. The following mechanism, including close request and fin exchange, support the reliable data transport and graceful connection closing services.

6.1.12.1 Close Requests A close request indicates that the local ULP has completed its data transfer over the connection. A ULP may close a connection at any time on its own initiative. Closing connections is intended to be a graceful operation in the sense that outstanding send requests will be transmitted (and retransmitted), as flow control permits, until all have been serviced. Thus, it should be acceptable to make several send requests, followed by a close request, and expect all the data to be sent to the destination ULP.

It should also be clear that ULPs should continue to accept data on closing connections, since the other ULP may be trying to transmit the last of its data. Thus, a close request means *"I have no more to send" but does not mean* "I will not receive any more." It may happen (if the upper level protocol is not well thought out) that the closing side is unable to get rid of all its data before timing out. In this event, a close turns into abort request, and the closing TCP gives up.

Because closing a connection requires communication with the foreign TCP, connections may remain in the closing state for a short time. Attempts to reopen the connection before the TCP replies to the close request will result in error responses.

A close service request also implies the push function.

6.1.12.2 FIN Exchange Examples The fin control flag in the segment header is exchanged with the same synchronization mechanism, the three-way handshake, used for connection opening. From the TCP entity perspective, there are essentially three cases for FIN exchange. One, the local ULP initiates connection closing with a CLOSE service request. Two, the remote TCP entity sends a FIN segment indicating that the remote ULP has issued a close request. Three, both ULPs simultaneously issue close requests.

Case 1: Local ULP Initiates Connection Close In this case, a FIN segment can be constructed and placed on the outgoing segment queue. No further send requests from the ULP will be accepted by the TCP, and it enters the FIN-WAIT-1 state. All segments preceding and including FIN will be retransmitted until acknowledged. When the other TCP has both acknowledged the FIN and sent a FIN of its own, the first TCP can ACK this FIN. Note that a TCP receiving a

FIN will ACK but not send its own FIN until its ULP has closed the connection also.

	TCP A		TCP B
1.	ESTABLISHED		ESTABLISHED
2.	(Close) FIN-WAIT-1	--> <SEQ=100><ACK=300><CTL=FIN,ACK>	--> CLOSE-WAIT
3.	FIN-WAIT-2	<-- <SEQ=300><ACK=101><CTL=ACK>	<-- CLOSE-WAIT
4.	TIME-WAIT	<-- <SEQ=300><ACK=101><CTL=FIN,ACK>	<-- (Close) LAST-ACK
5.	TIME-WAIT	--> <SEQ=101><ACK=301><CTL=ACK>	--> CLOSED
6.	(2 MSL) CLOSED		

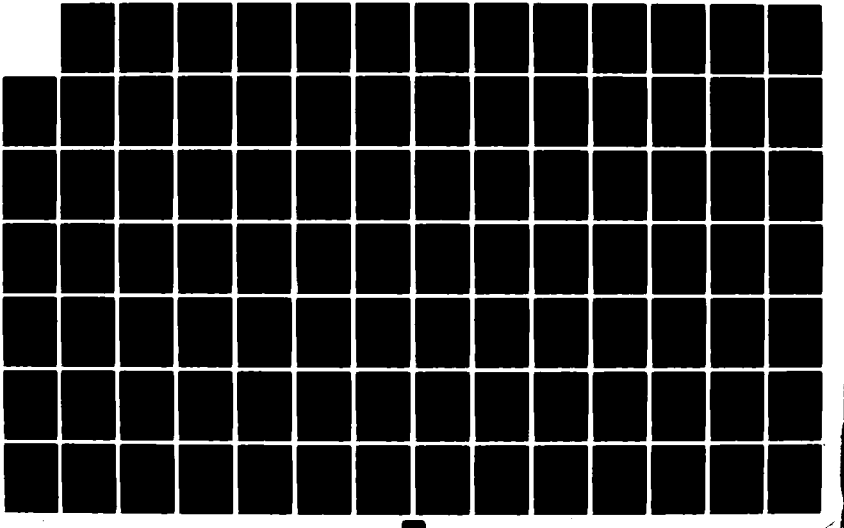
Normal Close Sequence

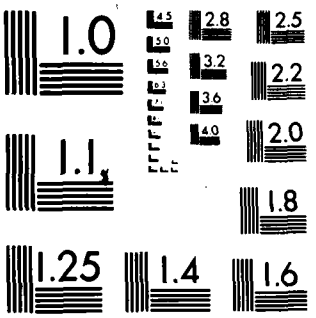
Case 2: TCP Receives FIN from Remote TCP If an unsolicited FIN arrives from the network, the receiving TCP can ACK it and tell the ULP that the connection is closing. The ULP will respond with a close request, upon which the TCP can send a FIN to the other TCP after sending any remaining data. The TCP then waits until its own FIN is acknowledged whereupon it deletes the connection. If an ACK is not forthcoming, after the ULP timeout the connection is aborted and the ULP is informed.

Case 3: ULPs Close Simultaneously Simultaneous close requests by both ULPs at each end of a connection cause FIN segments to be exchanged. When all segments preceding the FINs have been processed and acknowledged, each TCP can ACK the FIN it has received. Both will, upon receiving these ACKs, delete the connection.

AD-A126 559 PROPOSED DOD (DEPARTMENT OF DEFENSE) TRANSMISSION CONTROL PROTOCOL STANDARD(U) SYSTEM DEVELOPMENT CORP SANTA MONICA CA 06 JUL 82 SDC-TM-7172/482/00 3/3

UNCLASSIFIED DCA100-82-C-0036 F/G 17/2 NL





```
    TCP A                                                    TCP B

1.  ESTABLISHED                                              ESTABLISHED

2.  (ULP A issues CLOSE)                                 (ULP B issues CLOSE)
    FIN-WAIT-1  --> <SEQ=100><ACK=300><CTL=FIN,ACK>  ... FIN-WAIT-1
                <-- <SEQ=300><ACK=100><CTL=FIN,ACK>  <--
                ... <SEQ=100><ACK=300><CTL=FIN,ACK>  -->

3.  CLOSING     --> <SEQ=101><ACK=301><CTL=ACK>      ... CLOSING
                <-- <SEQ=301><ACK=101><CTL=ACK>      <--
                ... <SEQ=101><ACK=301><CTL=ACK>      -->

4.  TIME-WAIT                                                TIME-WAIT
    (2 MSL)                                                  (2 MSL)
    CLOSED                                                   CLOSED
```

Simultaneous Close Sequence

6.1.12.3 Quiet Time Concept While the clock-based ISN generation prevents overlap of sequence number use under normal conditions, special measures must be taken in situations where a host crashes (or restarts), resulting in a TCP's loss of knowledge concerning the sequence numbers in use on active connections, and the current ISN value. After crash recovery, a TCP may create segments containing the same or overlapping sequence numbers as those in pre-crash connection incarnations, causing confusion and misdelivery at the receiver. Even hosts managing to remember the time of day used as a basis for ISN selection are not immune to this problem, as the following example illustrates:

> Suppose, for example, that a connection is opened starting with sequence number S. Suppose that this connection is not heavily used and that eventually the initial sequence number function (ISN(t)) takes on a value equal to the sequence number, say S1, of the last segment sent by this TCP on a particular connection. Now suppose, at this instant, the host crashes, recovers, and establishes a new incarnation of the connection. The initial sequence number chosen is S1 = ISN(t) -- last used sequence number on the old incarnation of the connection!! If the recover occurs quickly enough, any old duplicates in the network bearing sequence numbers in the neighborhood of S1 may arrive and be accepted as new packets by the receiver of the new incarnation of the connection.

The problem is that the recovering host may not know for how long it crashed nor does it know whether there are still old duplicates in the system form earlier connection incarnations.

One way to handle these situations is to require that a TCP must "keep quiet", that is, refrain from emitting segments, for a maximum segment lifetime (MSL) before assigning any sequence numbers. This quiet time restriction allows the segments from earlier connection incarnations to drain from the network.

For this specification, the MSL is assumed to be 2 minutes. This is an engineering choice, and may be changed as experience dictates. TCP implementors violating this restriction run the risk of causing some old data to be accepted as new or new data rejected as old duplicates. Note that if a TCP is reinitialized yet retains its knowledge of sequence numbers in use, the quiet time restriction does not apply; however, care should be taken to use sequence numbers larger than those recently used.

6.1.13 Resets

One of the control flags of the TCP header is the reset flag. A segment carrying a reset flag set true is called a reset. Resets are used to to abruptly close established connections, refuse connection attempts, and respond to segments apparently not intended for the current incarnation of a connection. The following paragraphs define the rules for reset generation and for reset validation and processing.

6.1.13.1 Reset Generation Each paragraph below specifies when a reset should be sent, the sequence number and, when needed, the acknowledgement number necessary to make the reset segment acceptable to the remote TCP.

When either ULP of the communicating ULP-pair issues an Abort service request, its local TCP informs the remote TCP with a reset segment carrying a sequence number field equal to SEND_NEXT.

As a general rule, reset (RST) must be sent whenever a segment arrives which apparently is not intended for the current connection. A reset must not be sent if it is not clear that this is the case. Specific examples of reset generation in response to misdirected segments are presented in three groups of states:

1. When the connection does not exist (i.e., its state is CLOSED) then a reset is sent in response to any incoming segment except another reset. In particular, SYNs addressed to nonexistent connections are rejected in this manner.

   If such an incoming segment has an ACK field, the reset segment takes its sequence number from the ACK field of the incoming segment; otherwise, the reset segment takes a sequence number value of zero and an acknowledgement number equal to the sum of the sequence number and text length of the incoming segment. The connection remains in the CLOSED state.

2. When the connection is in any nonsynchronized state (LISTEN, SYN-SENT, SYN-RECEIVED), a reset is sent in the following cases: The incoming segment acknowledges something not yet sent (that is, the segment carries an unacceptable ACK), or an incoming segment carries security information which does not exactly match that designated for the connection.

   Resets generated in the nonsynchronized states are made acceptable as

follows. When the incoming segment has an ACK field, the reset segment takes its sequence number from the ACK field of the incoming segment; otherwise, the reset segment carries a sequence number equal to zero and and acknowledgement field set to the sum of the sequence number and text length of the incoming segment. The connection remains in the same state.

3. If the connection is in a synchronized state (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), any unacceptable segment (such as one with an out of window sequence number or an unacceptable acknowledgement number) must elicit only an empty acknowledgement segment containing the current send sequence number (SEND_NEXT) and an acknowledgement indicating the next sequence number expected to be received (RECV_NEXT). (Note that if the unacceptable segment is an empty ACK segment, replying with an ACK may result in a cascade of ACKs. In general, don't ACK an unacceptable empty ACK segment.) The connection remains in the same state.

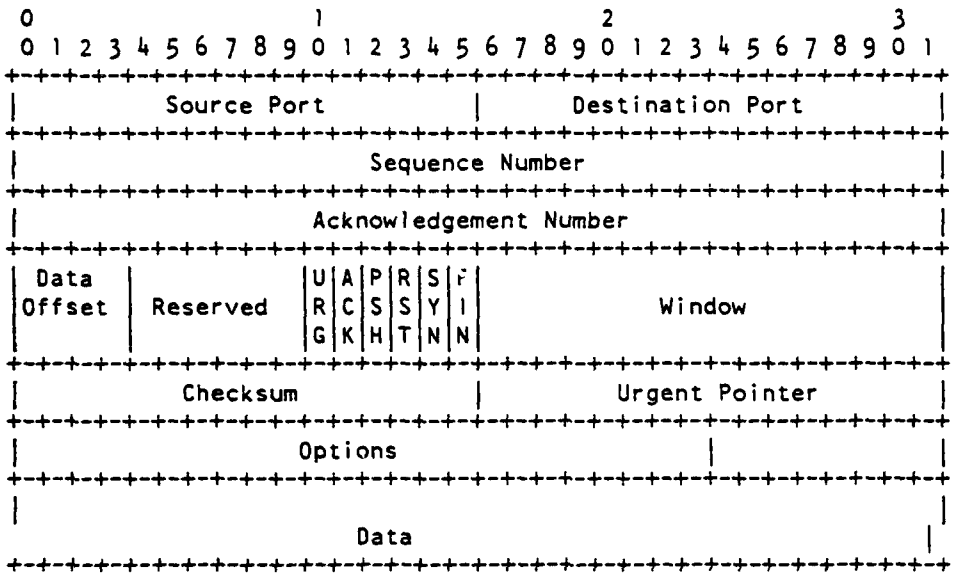
   If an incoming segment has security information or a precedence level which does not exactly match those designated for the connection, a reset is sent; the connection enters the CLOSED state. The reset segment takes its sequence number from the ACK field of the incoming segment.

6.1.13.2 Reset Processing In all states except SYN-SENT, all reset (RST) segments are validated by checking their sequence number fields. A reset is valid if its sequence number is in the connection's receive window. In the SYN-SENT state (a RST received in response to an initial SYN), the RST is valid if the ACK field acknowledges the SYN.

The receiver of a RST first validates it, then changes state. If the receiver was in the LISTEN state, it ignores it. If the receiver was in SYN-RECEIVED state and had previously been in the LISTEN state, then the receiver returns to the LISTEN state; otherwise, the receiver aborts the connection and goes to the CLOSED state. If the receiver was in any other state, it aborts the connection and advises the ULP and goes to the CLOSED state.

6.2 TCP HEADER FORMAT

A summary of the contents of a TCP header follows:



TCP Header Format

Note that each tick mark represents one bit position. Each field description below includes its name, an abbreviation, and the field size. Where applicable, the units, the legal range of values, and a default value appears.

6.2.1 Source Port

abbrev: SRC PORT field size: 16 bits

The source port number.

6.2.2 Destination Port

abbrev: DEST PORT field size: 16 bits

The destination port number.

6.2.3 Sequence Number

abbrev: SEQ field size: 32 bits
units : octets range : 0 - 2**32-1

Usually, this value represents the sequence number of the first data octet of a segment. However, if a SYN is present, the sequence number is the initial

sequence number (ISN) covering the SYN; the first data octet is then numbered ISN+1.

6.2.4 Acknowlegement Number
abbrev: ACK field size: 32 bits
units : octets range : 0 - 2**32-1

If the ACK control bit is set this field contains the value of the next sequence number that the sender of the segment is expecting to receive.

6.2.5 Data Offset
abbrev: none field size: 4 bits
units : 32-bits range : 5 - 15 default : 5

This field indicates the number of 32 bit words in the TCP header. From this value, the beginning of the data can be computed. The TCP header (even one including options) is an integral number of 32 bits long.

6.2.6 Reserved
abbrev: none field size: 6 bits

Reserved for future use. Must be set to zero.

6.2.7 Control Flags
abbrev: below field size: 6 bits(from left to right)

URG: Urgent Pointer field significant
ACK: Acknowledgment field significant
PSH: Push Function
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: No more data from sender

These flags carry control information used for connection establishment, connection termination, and connection maintenance.

6.2.8 Window
abbrev: WNDW field size: 2 octets
units : octets range : 0 - 2**16-1 default : none

The number of data octets beginning with the one indicated in the acknowledgment field which the sender of this segment is willing to accept.

6.2.9 Checksum
abbrev: none field size: 2 octets

The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header and text. The checksum also covers a 96 bit pseudo header conceptually prefixed to the TCP header. This pseudo header contains the Source Address, the Destination Address, the Protocol, and TCP length. The checksum algorithm is defined in section 6.1.5.

6.2.10 Urgent Pointer
abbrev: URGPTR field size: 2 octets
units : octets range : 0 - 2**16-1 default : 0

This field indicates the current value of the urgent pointer as a positive offset from the sequence number in this segment. The urgent pointer points to the sequence number of the octet following the urgent data. This field is only be interpreted in segments with the URG control bit set.

6.2.11 Options
abbrev: OPT field size: variable

If present, options occupy space at the end of the TCP header and are a multiple of 8 bits in length. All options are included in the checksum. An option may begin on any octet boundary.

There are two cases for the format of an option:

1. A single octet of option-kind.

2. An octet of option-kind, an octet of option-length, and the actual option-data octets.

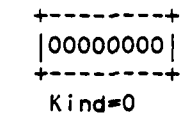
The option-length counts the two octets of option-kind and option-length as well as the option-data octets. Note that the list of options may be shorter than the data offset field might imply. The content of the header beyond the End-of-Option option must be header padding (i.e., zero).

Currently defined options include (kind indicated in octal):

Kind	Length	Meaning
0	-	End of option list.
1	-	No-Operation.
2	4	Maximum Segment Size.

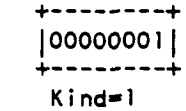
6.2.11.1 Specific Option Definitions

6.2.11.1.1 End of Option List



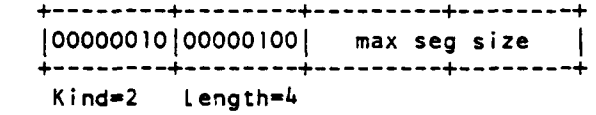
This option code indicates the end of the option list. This might not coincide with the end of the TCP header according to the Data Offset field. This is used at the end of all options, not the end of each option, and need only be used if the end of the options would not otherwise coincide with the end of the TCP header.

6.2.11.1.2 No-Operation



This option code may be used between options, for example, to align the beginning of a subsequent option on a word boundary. There is no guarantee that senders will use this option, so receivers must be prepared to process options even if they do not begin on a word boundary.

6.2.11.1.3 Maximum Segment Size



If this option is present, then it communicates the maximum receive segment size at the TCP which sends this segment. This field must only be sent in the initial connection request (i.e., in segments with the SYN control bit set). If this option is not used, any segment size is allowed.

6.2.12 Padding

abbrev: none field size: variable

The padding is used to ensure that the TCP header ends and data begins on a 32 bit boundary. The padding is composed of zeros.

6.3 EXTENDED STATE MACHINE SPECIFICATION OF TCP ENTITY

The TCP protocol entity is specified with an extended state machine made up a of set of states, a set of transitions between states, and a set of input events causing the state transitions. The following specification is made up of a machine instantiation identifier, a state diagram, a state vector, data structures, an event list, and a correspondence between events and actions. In addition, an extended state machine has an initial state whose value are assumed at state machine instantiation.

6.3.1 Machine Instantiation Identifier

One state machine instance exists for each connection. A connection, and hence a state machine, is uniquely named by either of the two machine instantiation identifiers that exist: the socket pair and the local connection name.

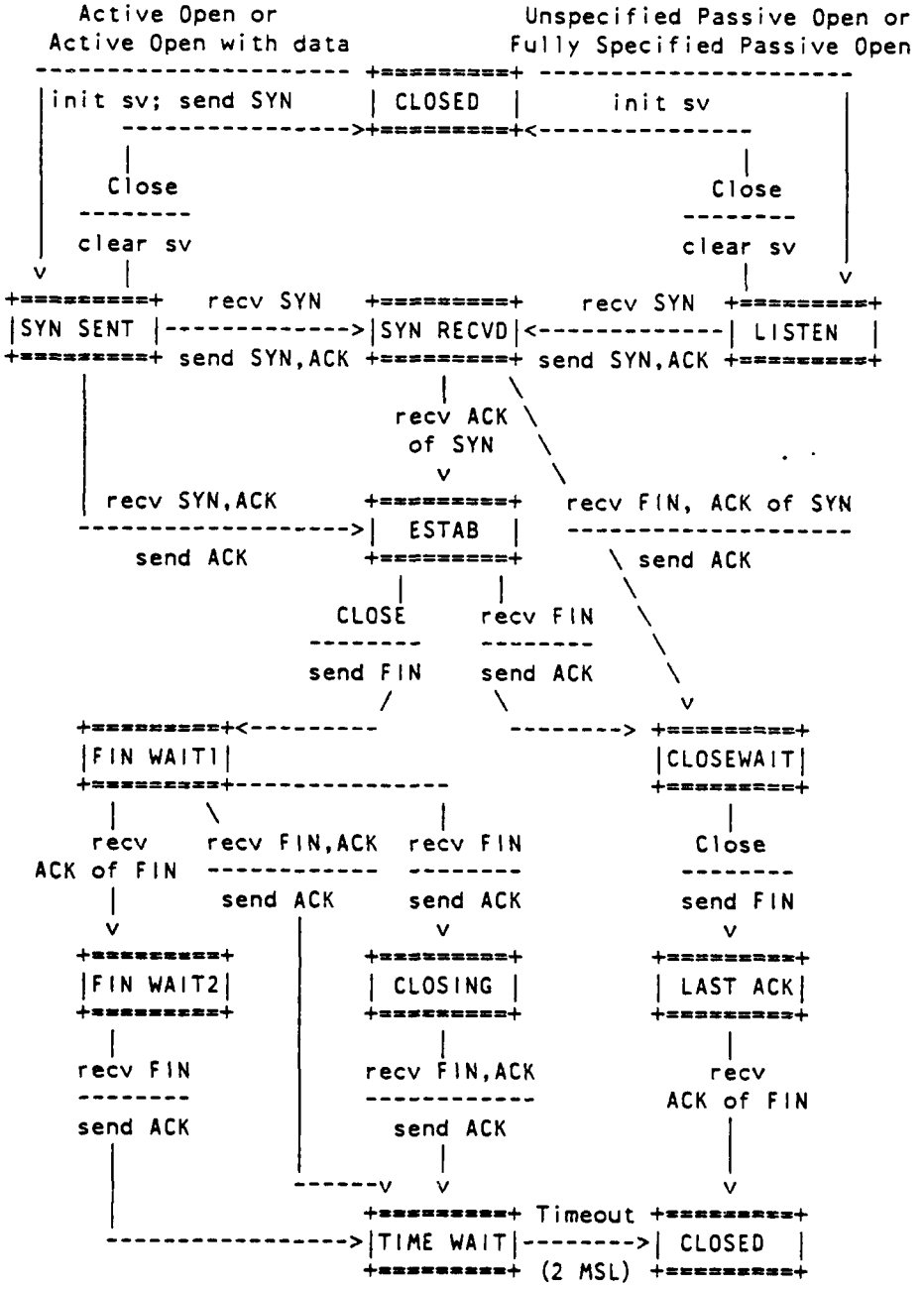
6.3.1.1 Socket Pair Identifier TCP segments delivered by the network and connection establishment service requests (Active Open, Active Open with Data, Full Passive Open, and Unspecified Passive Open) carry and thus are bound to a connection with the following values:

- o source address
- o source port
- o destination address
- o destination port

6.3.1.2 Local Connection Name A TCP entity assigns an identifier, a local connection name, that appears in all service responses and all service requests except for active and passive open requests.

6.3.2 State Diagram

The following diagram summarizes the state machine for the TCP entity:



TCP ENTITY STATE SUMMARY

LEGEND

recv - NET_DELIVER of segment
send - NET_SEND of segment
sv - state vector
init - initialize

2 MSL - 2 max segment lifetimes clear - nullify

Please note the diagram is intended only as a summary and does not supersede the formal definition that follows.

6.3.3 State Vector

The elements comprising the state vector of a TCP entity appear below. Each element name is followed by by the name of the corresponding record element in the state vector structure "sv" declared in Section 6.3.4.1.

1. state name (sv.state): the current state of the entity state machine from the following list: CLOSED, LISTEN, SYN_RECVD, SYN_SENT, ESTAB, FIN_WAIT1, FIN_WAIT2, CLOSE_WAIT, CLOSING, LAST_ACK, TIME_WAIT.

2. source address (sv.source_addr): the internet address naming the location of the local ULP.

3. source port (sv.source_port): the identifier of the local ULP.

4. destination address (sv.destination_addr): the internet address of the location of the the ULP at the other end of the connection.

5. destination port (sv.destination_port): the identifier of the ULP at the other end of the connection.

6. lcn (sv.lcn): local connection name, the identifier associated with this end of the connection.

7. open mode (sv.open_mode): the type of open request issued by the local ULP, either ACTIVE or PASSIVE.

8. original precedence (sv.original_prec): one of eight levels of special handling requested by the local ULP in the open request.

9. actual precedence (sv.actual_prec): one of eight levels of special handling negotiated during connection establishment and verified throughout connection lifetime.

10. security (sv.sec): information (including security level, compartment, handling restrictions, and transmission control code) defined by the local ULP.

11. ulp timeout (sv.ulp_timeout): the maximum delay allowed for data transmitted on the connection.

12. send unacknowledged sequence number (sv.send_una): oldest unacknowledged send sequence number (i.e. left edge of send window).

13. send next sequence number (sv.send_next): sequence number of the next data octet to be sent.

14. send free sequence number (sv.send_free): sequence number of the first free octet in the send queue (i.e. the next octet to be received from the local ULP).

15. send window (sv.send_wndw): allowed number of octets that may be sent to the remote TCP relative to the send unacknowledged sequence number.

16. send urgent sequence number (sv.send_urg): sequence number of the last octet of urgent data in send stream.

17. send push sequence number (sv.send_push): sequence number of the last octet of pushed data in the send stream.

18. send last window update 1 (sv.send_lastup1): sequence ıber of the incoming segment used for last window update.

19. send last window update 2 (sv.send_lastup2): acknowledgmer ..mber of the incoming segment used for last window update.

20. send initial sequence number (sv.send_isn): sequence numb the original SYN sent.

21. send fin flag (sv.send_finflag): indicates that the local ULP has issued a Close request.

22. send maximum segment size (sv.send_max_seg): maximum sized segment to be sent to the remote TCP on this connection.

23. send queue (sv.send_queue): location of data received from the local ULP and either awaiting acknowledgement, or awaiting transmission. This area is accessed only by the data management routines.

24. receive next sequence number (sv.recv_next): sequence number of next data octet expected to be received.

25. receive save sequence number (sv.recv_next): sequence number of next data octet to be delivered to the local ULP.

26. receive window (sv.recv_wndw): allowed number of data octets to be received from the remote TCP starting with the receive next sequence number.

27. receive alloc (sv.recv_alloc): the number of data octets will to be accepted by the local ULP.

28. receive urgent sequence number (sv.recv_urg): sequence number of the last octet of urgent data in receive stream.

29. receive push sequence number (sv.recv_push): sequence number of the last octet of pushed data in receive stream.

30. receive initial sequence number (sv.recv_isn): sequence number of the SYN received from remote TCP.

31. receive fin flag (sv.recv_finflag): indicates that fin has been received from the remote TCP.

32. receive queue (sv.recv_queue): location of data accepted from remote TCP before delivery to local ULP. This area is accessed only by the data management routines.

6.3.4 Data Structures

The TCP entity state machine references certain data areas corresponding to the state vector, the service requests and responses on the upper interface, and the service requests and responses on the lower interface. For clarity in the events and actions section, these data structures are declared in ADA. However, a data structure may may partially typed or untyped where specific formats or data types are implementation dependent.

6.3.4.1 state vector The TCP entity state vector is defined in Section 6.3.1 above. The corresponding structure is declared as:

```
sv : state_vector_type;

type state_vector_type is
  record
     state : (CLOSED, LISTEN, SYN_RECVD, SYN_SENT
              ESTAB, FIN_WAIT1, FIN_WAIT2,
              CLOSE_WAIT,CLOSING, LAST_ACK, TIME_WAIT);
     source_addr : address_type;
     source_port : TWO_OCTETS;
     destination_addr : address_type;
     destination_port : TWO_OCTETS;
     lcn : integer;
     open_mode : (ACTIVE, PASSIVE);
     original_prec : precedence_type;
     actual_prec : precedence_type;
     sec : security_type;
     ulp_timeout : integer;
     send_una : sequence_number_type;
     send_next :  sequence_number_type;
     send_free :  sequence_number_type;
     send_wndw : integer;
     send_urg  :  sequence_number_type;
     send_push :  sequence_number_type;
     send_lastup1 :  sequence_number_type;
     send_lastup2 :  sequence_number_type;
     send_isn  :  sequence_number_type;
     send_finflag : boolean;
     send_max_seg : integer;
     send_queue : queue_type;
     recv_next :  sequence_number_type;
     recv_save :  sequence_number_type;
```

```
        recv_wndw : integer;
        recv_alloc : integer;
        recv_urg  :  sequence_number_type;
        recv_push :  sequence_number_type;
        recv_isn  :  sequence_number_type;
        recv_finflag : boolean;
        recv_queue : queue_type;
      end record;
```

6.3.4.2 from ULP The from_ULP structure holds the service request parameters and data associated with the service request primitives as specified in Section 3.1.1. The from_ULP structure is declared as:

```
type from_ULP_type is
  record
    request_name : (Unspecified_Passive_Open, Full_Passive_Open,
                    Active_Open, Active_Open_with_data,
                    Send, Allocate, Close, Abort, Status);
    source_addr
    source_port
    destination_addr
    destination_port
    lcn
    ulp_timeout
    precedence
    security
    data
    data_length
    push_flag
    urgent_flag
  end record;
```

6.3.4.3 to ULP The to_ULP structure holds service response parameters and data as specified in Section 3.1.2. Although the structure is composed of the parameters from all the service requests, a particular service response will use only those structure elements corresponding to its specified parameters. This structure directly corresponds to the to_ULP structure declared in 3.3.4.3 of the upper service specification. The to_ULP structure is declared as:

```
type to_ULP_type is
  record
    service response : (OPEN_ID, OPEN_FAIL, OPEN_SUCCESS,
                        DELIVER, CLOSING, TERMINATE, ERROR);
    source_addr
    source_port
    destination_addr
    destination_port
    lcn
    data
```

```
        data_length
        urgent_flag
        error_desc
        status_block : status_block_type;
      end record;

    type status_block_type is
      record
         connection_state
         send_window
         receive_window
         amount_of_unacked_data
         amount_of_unreceived_data
         urgent_state
         precedence
         security
         ulp_timeout
      end record;
```

6.3.4.4 to_NET The to_NET structure holds the service request parameters and data associated with the NET_SEND service request specified in Section 5.1.1. This structure directly corresponds to the to_NET structure declared in 5.2.2 of the lower layer service requirements section. The to_NET structure is declared as:

```
    type to_NET_type is
      record
        source_addr
        destination_addr
        protocol
        type_of_service is
          record
              precedence
              reliability
              delay
              throughput
              reserved
          end record;
        time_to_live
        dont_fragment
        length
        seg : segment_type;
        options
      end record;
```

6.3.4.5 from_NET The from_NET structure holds the service response parameters and data associated with the NET_DELIVER service response, as specified in Section 5.1.2. This structure directly corresponds to the from_NET structure declared in 5.2.3 of the lower layer service requirements section. The from_NET structure is declared as:

```
type from_NET_type is
  record
    source_addr
    destination_addr
    protocol
    type_of_service is
      record
          precedence
          reliability
          delay
          throughput
          reserved
      end record;
    length
    seg : segment_type;
    options
    error
  end record;
```

6.3.4.6 segment type A segment_type structure holds a TCP segment made up of a header portion and a data portion as specified in Section 6.2. A segment_type structure is declared as:

```
type segment_type is
  record
     source_port      : TWO_OCTETS;
     destination_port : TWO_OCTETS;
     seq_num    : FOUR_OCTETS;
     ack_num    : FOUR_OCTETS;
     data_offset : HALF_OCTET;
     reserved   : SIX_EIGHTHS_OCTET;
     urg_flag   : ONE_BIT;
     ack_flag   : ONE_BIT;
     push_flag  : ONE_BIT;
     rst_flag   : ONE_BIT;
     syn_flag   : ONE_BIT;
     fin_flag   : ONE_BIT;
     wndw       : TWO_OCTETS;
     checksum   : TWO_OCTETS;
     urgptr     : TWO_OCTETS;
     options    : is array of OCTET;
     padding    : is array of OCTET;
     data       : is array of OCTET;
  end record;
```

6.3.4.7 Supplemental Type Declarations

```
type address_type is FOUR_OCTETS;
type sequence_number_type is FOUR_OCTETS;
type precedence_type is INTEGER range 0..7;
type security_type is
```

```
      record
         security_level : HALF_OCTET;
         compartment    : TWO_OCTETS;
         handling       : TWO_OCTETS;
         trans_control_code : THREE_OCTETS;
      end record;

   subtype OCTET is INTEGER range 0..255;
   subtype HALF_OCTET is INTEGER range 0..15;
   subtype FIVE_EIGHTHS_OCTET is INTEGER range 0..31;
   subtype TWO_OCTETS is INTEGER range 0..2**16-1;
   subtype THREE_OCTETS is INTEGER range 0..2**24-1;
   subtype FOUR_OCTETS is INTEGER range 0..2**32-1;

   NULL_RESERVED      : constant FIVE_EIGHTHS_OCTET := 0;
   OPTIONLESS_HEADER  : constant INTEGER := 5;
   NORMAL             : constant INTEGER := 0;
   NULL               : constant INTEGER := 0;
      --NULL assumed to be outside the sequence number space.
   DEFAULT_PRECEDENCE : constant INTEGER := 0;
   DEFAULT_SECURITY   : constant INTEGER := 0;
   DEFAULT_TIMEOUT    : constant INTEGER := 01111000(8);
   ONE_MINUTE_TTL     : constant INTEGER := 00111100(8);
   THIS_ADDRESS       : constant INTEGER;   --impl. dependent
   TCP_ID             : constant INTEGER;   --reference [5]
```

6.3.5 Event List

The events for the TCP entity state machine are drawn from the service request primitives defined in the service definition of Section 3.1. Optional service request parameters are shown in brackets. The capitalized list of parameters represent the actual values of the parameters passed by the service primitive. The event list:

```
1. Unspecified Passive Open( SOURCE_PORT,
                             [,TIMEOUT] [,PRECEDENCE] [,SECURITY] );

2. Full Passive Open( SOURCE_PORT,
                      DESTINATION_PORT, DESTINATION_ADDRESS,
                      [,TIMEOUT] [,PRECEDENCE] [,SECURITY] );

3. Active Open( SOURCE_PORT,
                DESTINATION_PORT, DESTINATION_ADDRESS
                [,TIMEOUT] [,PRECEDENCE] [,SECURITY] );

4. Active Open w/data( SOURCE_PORT,
                       DESTINATION_PORT, DESTINATION_ADDRESS
                       [,TIMEOUT] [,PRECEDENCE] [,SECURITY] );
                       DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG );

5. Send( LCN, DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG [,TIMEOUT] );
```

6. Allocate(LCN, DATA_LENGTH)

7. Close(LCN)

8. Abort(LCN)

9. Status(LCN)

10. NET_DELIVER(SOURCE_ADDRESS, DESTINATION_ADDRESS, PROTOCOL,
TOS[precedence, reliability, delay, throughput],
OPTIONS[security], LENGTH, DATA)

11. Retransmission Timeout

12. ULP Timeout

13. Time Wait Timeout

6.3.6 Events and Actions

This section is organized in three parts. The first part contains a decision table representation of state machine events and actions. The decision tables are organized by state; each table corresponds to one event.

The second part specifies the decision functions appearing at the top of each column of a decision table. These functions examine attributes of the event and the state vector to return a set of decision results. The results become the elements of each column.

The third part specifies action procedures appearing at the right of every row. Each row of the decision table combines the decision results to determine appropriate event processing. These procedures specify event processing algorithms in detail.

6.3.6.1 Decision Tables The Status event can occur in any state except closed; TCP's action is to return the current state_vector information as specified in the STATUS_RESPONSE service response.

If the primary state vector element is not changed in the decision table row corresponding to an event, the "primary" state remains unchanged.

The checksum is assumed to be computed for all incoming segments. When the computed checksum does not match the segment's header checksum field, the segment is discarded without being acknowledged.

STATE = CLOSED

Legend
d = "don't care" condition

Event: Active Open(LOCAL_PORT, REMOTE_PORT, REMOTE_ADDRESS
[TIMEOUT] [PRECEDENCE] [SECURITY])

Actions:

Resources suffic open?	Sec prec allowed	
NO	d	error("Insufficient resources.")
YES	NO	error("Security/precedence not allowed.")
YES	YES	open;gen_syn(ALONE);sv.state=SYN_SENT

Event: Active Open with Data(LOCAL_PORT, REMOTE_PORT, REMOTE_ADDRESS,
[TIMEOUT] [PRECEDENCE] [SECURITY]
DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG)

Actions:

Resources suffic open?	Sec prec allowed	
NO	d	error("Insufficient resources.")
YES	NO	error("Security/precedence not allowed.")
YES	YES	open;gen_syn(WITH_DATA);sv.state=SYN_SENT

Event: Full Passive Open(LOCAL_PORT, REMOTE_PORT, REMOTE_ADDRESS,
[TIMEOUT] [PRECEDENCE] [SECURITY])

Actions:

Resources suffic open?	Sec prec allowed	
NO	d	error("Insufficient resources.")
YES	NO	error("Security/precedence not allowed.")
YES	YES	open; sv.state=LISTEN

STATE = CLOSED (con't)

Event: Unspecified Passive Open(LOCAL_PORT, [TIMEOUT] [PRECEDENCE] [SECURITY])

Actions:

Resources suffic open?	Sec prec allowed	
NO	d	error("Insufficient resources.")
YES	NO	error("Security/precedence not allowed.")
YES	YES	open; sv.state=LISTEN

Event: Send()
or Close()
or Abort()
or Allocate()

Actions: error("Connection does not exist.")

Event: NET_DELIVER(SOURCE ADDRESS, DESTINATION ADDRESS, PROTOCOL, TOS[precedence, reliability, delay, throughput], OPTIONS[security], LENGTH, DATA)

Actions:

RST on?	ACK on?	
NO	NO	reset(SEG)
NO	YES	reset(SEG)
YES	d	-- no action

==========
STATE = LISTEN
==========

Event: Close(LCN)
or Abort(LCN)

Actions: reset_self(UC); sv.state=CLOSED

Event: Allocate(LCN, DATA_LENGTH)

Actions: new_allocation

Event: Send()

Actions: error("Illegal request.")

Event: Active Open()
or Active Open with data()
or Full Passive Open()
or Unspecified Passive Open()

Actions: error("Connection already exists.")

Event: NET_DELIVER(SOURCE ADDRESS, DESTINATION ADDRESS, PROTOCOL,
TOS[precedence, reliability, delay, throughput],
OPTIONS[security], LENGTH, DATA)

Actions:

RST on?	ACK on?	SYN on?	Sec match?	sv prec vs seg prec	
NO	NO	NO	d	d	-- no action
NO	NO	YES	NO	d	reset(SEG)
NO	NO	YES	YES	GREATER or EQUAL	record_syn;gen_syn(WITH_ACK) sv.state=SYN_RECVD
NO	NO	YES	YES	LESS	record_syn; raise_prec; gen_syn(WITH_ACK);sv.state=SYN_RECVD
NO	YES	d	d	d	reset(SEG)
YES	d	d	d	d	-- no action

```
================
STATE = SYN_SENT
================

Event: Close( LCN )
       or Abort( LCN )

Actions: reset_self(UC); sv.state=CLOSED


Event: Send( LCN, DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG [TIMEOUT] )

Actions:
===========
|Resources|
| suffic  |
|  send?  |
===========
|   NO    |error("Insufficient resources.")
-----------------------------------------------
|   YES   |save_send_data
-----------------------------------------------


Event: Allocate( LCN, DATA_LENGTH )

Actions: new_allocation


Event: Active Open()
       or Active Open with data()
       or Full Passive Open()
       or Unspecified Passive Open()

Actions: error("Connection already exists.")


Event: Retransmisson Timeout

Actions: retransmit


Event: ULP Timeout

Actions: open_fail; sv.state=CLOSED
```

STATE = SYN_SENT (con't)

Legend
d = "don't care" condition

Event: NET_DELIVER(SOURCE ADDRESS, DESTINATION ADDRESS, PROTOCOL,
TOS[precedence, reliability, delay, throughput],
OPTIONS[security], LENGTH, DATA)

Actions:

ACK status test1	RST on?	Sec match?	sv prec vs seg prec	SYN on?	FIN on?	
NONE	NO	NO	d	d	d	reset(SEG)
NONE	NO	YES	d	NO	d	-- no action
NONE	NO	YES	GREATER or EQUAL	YES	NO	record_syn; send_ack(sv.recv_isn+1) sv.state=SYN_RECVD
NONE	NO	YES	GREATER or EQUAL	YES	YES	record_syn; send_ack(sv.recv_isn+1) save_fin; sv.state=SYN_RECVD
NONE	NO	YES	LESS	YES	NO	record_syn; raise_prec; send_ack(sv.recv_isn+1) sv.state=SYN_RECVD
NONE	NO	YES	LESS	YES	YES	record_syn; raise_prec send_ack(sv.recv_isn+1);save_fin sv.state=SYN_RECVD
NONE	YES	d	d	d	d	-- no action
INVAL	NO	d	d	d	d	reset(SEG)
INVAL	YES	d	d	d	d	-- no action
VALID	NO	NO	d	d	d	reset(SEG)
VALID	NO	YES	GREATER	d	d	reset(SEG)
VALID	NO	YES	LESS or EQUAL	NO	d	-- no action
VALID	NO	YES	LESS or EQUAL	YES	NO	raise_prec;conn_open sv.state=ESTAB
VALID	NO	YES	LESS or EQUAL	YES	YES	raise_prec; conn_open set_fin;sv.state=CLOSE_WAIT
VALID	YES	d	d	d	d	openfail;sv.state=CLOSED

```
==================
STATE = SYN_RECVD
==================

Event: Close( LCN )

Actions: send_fin; sv.state=FIN_WAIT1


Event: Abort( LCN )

Actions: reset(CURRENT); reset_self(UA); sv.state=CLOSED


Event: Send( LCN, DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG [TIMEOUT] )

Actions:
===========
|Resources|
| suffic  |
|  send?  |
===========
|   NO    | error("Insufficient resources.")
-----------------------------------------------
|   YES   | save_send_data
-----------------------------------------------


Event: Allocate( LCN, DATA_LENGTH )

Actions: new_allocation


Event: Active Open()
       or Active Open with data()
       or Full Passive Open()
       or Unspecified Passive Open()

Actions: error("Connection already exists.")


Event: Retransmission Timeout

Actions: retransmit


Event: ULP Timeout

Actions: reset(CURRENT); openfail; sv.state=CLOSED
```

STATE = SYN_RECVD (con't)

Legend
d = "don't care" condition

Event: NET_DELIVER(SOURCE ADDRESS, DESTINATION ADDRESS, PROTOCOL, TOS[precedence, reliability, delay, throughput], OPTIONS[security], LENGTH, DATA)

Actions:

Seq# status ?	RST on?	Sec prec match?	Open Mode ?	SYN in wndow	ACK status test1	Zero recv wndow	FIN seen?	
INVAL	NO	d	d	d	d	d	d	send_ack(sv.recv_next)
INVAL	YES	d	d	d	d	d	d	-- no action
VALID	NO	NO	PASS	d	d	d	d	reset(SEG); part_reset sv.state=LISTEN
VALID	NO	NO	ACT	d	d	d	d	reset(SEG);openfail sv.state=CLOSED
VALID	NO	YES	d	NO	NONE	d	d	-- no action
VALID	NO	YES	d	NO	INVAL	d	d	reset(SEG)
VALID	NO	YES	d	NO	VALID	NO	NO	conn_open;sv.state=ESTAB
VALID	NO	YES	d	NO	VALID	NO	YES	conn_open; set_fin sv.state=CLOSE_WAIT
VALID	NO	YES	d	NO	VALID	YES	d	update; check_urg sv.state=ESTAB
VALID	NO	YES	d	YES	d	d	d	reset(SEG);openfail sv.state=CLOSED
VALID	YES	d	PASS	d	d	d	d	part_reset;sv.state=LISTEN
VALID	YES	d	ACT	d	d	d	d	openfail;sv.state=CLOSED

```
=============
STATE = ESTAB
=============

Event: Close( LCN )

Actions: send_fin; sv.state=FIN_WAIT1


Event: Abort( LCN )

Actions: reset(CURRENT); reset_self(UA); sv.state=CLOSED


Event: Send( LCN, DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG [TIMEOUT] )

Actions:
===========
|Resources|
| suffic  |
|  send?  |
===========
|   NO    |error("Insufficient resources.")
-----------------------------------------------
|   YES   |dispatch
-----------------------------------------------



Event: Allocate( LCN, DATA_LENGTH )

Actions: new_allocation


Event: Active Open()
       or Active Open with data()
       or Full Passive Open()
       or Unspecified Passive Open()

Actions: error("Connection already exists.")


Event: Retransmission Timeout

Actions: retransmit


Event: ULP Timeout

Actions: reset(CURRENT); reset_self(UT); sv.state=CLOSED
```

STATE = ESTAB (con't)

Legend
d = "don't care" condition

Event: NET_DELIVER(SOURCE ADDRESS, DESTINATION ADDRESS, PROTOCOL,
TOS[precedence, reliability, delay, throughput],
OPTIONS[security], LENGTH, DATA)

Actions:

Seq# status ?	RST on?	Sec prec match?	SYN in wndow	ACK status test2	Zero recv wndow	FIN seen?	
INVAL	NO	d	d	d	d	d	send_ack(sv.recv_next)
INVAL	YES	d	d	d	d	d	-- no action
VALID	NO	NO	d	d	d	d	reset(SEG);reset_self(SP) sv.state=CLOSED
VALID	NO	YES	NO	NONE	d	d	-- no action
VALID	NO	YES	NO	INVAL	d	d	send_ack(sv.recv_next)
VALID	NO	YES	NO	VALID	NO	NO	update;accept
VALID	NO	YES	NO	VALID	NO	YES	update;accept;set_fin sv.state=CLOSE_WAIT
VALID	NO	YES	NO	VALID	YES	d	update;check_urg
VALID	NO	YES	YES	d	d	d	reset(SEG);reset_self(SF) sv.state=CLOSED
VALID	YES	d	d	d	d	d	reset_self(RA);sv.state=CLOSED

```
==================
STATE = CLOSE_WAIT
==================

Event: Send( LCN, DATA, DATA_LENGTH, PUSH_FLAG, URGENT_FLAG [TIMEOUT] )

Actions:
===========
|Resources|
| suffic  |
|  send?  |
===========
|   NO    | error("Insufficient resources.")
-----------------------------------------------
|   YES   | dispatch
-----------------------------------------------


Event: Allocate( LCN, DATA_LENGTH )

Actions: new_allocation


Event: Active Open()
       or Active Open with data()
       or Full Passive Open()
       or Unspecified Passive Open()

Actions: error("Connection already exists.")


Event: Close( LCN )

Actions: send_fin; sv.state=LAST_ACK


Event: Abort( LCN )

Actions: reset(CURRENT); reset_self(UA); sv.state=CLOSED


Event: Retransmission Timeout

Actions: retransmit


Event: ULP Timeout

Actions: reset(CURRENT); reset_self(UT); sv.state=CLOSED
```

STATE = CLOSE_WAIT (con't)

Legend
d = "don't care" condition

Event: NET_DELIVER(SOURCE ADDRESS, DESTINATION ADDRESS, PROTOCOL,
TOS[precedence, reliability, delay, throughput],
OPTIONS[security], LENGTH, DATA)

Actions:

Seq# status ?	RST on?	Sec prec match?	SYN in wndow	ACK status test2?	
INVAL	NO	d	d	d	send_ack(sv.recv_next)
INVAL	YES	d	d	d	-- no action
VALID	NO	NO	d	d	reset(SEG);reset_self(SP);sv.state=CLOSED
VALID	NO	YES	NO	NONE	-- no action
VALID	NO	YES	NO	INVAL	send_ack(sv.recv_next)
VALID	NO	YES	NO	VALID	update
VALID	NO	YES	YES	d	reset(SEG);reset_self(SF);sv.state=CLOSED
VALID	YES	d	d	d	reset_self(RA); sv.state=CLOSED

```
===============
STATE = CLOSING
===============

Event: Allocate(LCN, DATA_LENGTH)

Actions: new_allocation


Event: Send()
       or Close()

Actions: error("Connection closing.")


Event: Active Open()
       or Active Open with data()
       or Full Passive Open()
       or Unspecified Passive Open()

Actions: error(Connection already exists.")


Event: Abort( LCN )

Actions: reset(CURRENT); reset_self(UA); sv.state=CLOSED;


Event: Retransmission Timeout

Actions. retransmit


Event: ULP Timeout

Actions: reset(CURRENT); reset_self(UT); sv.state=CLOSED
```

STATE = CLOSING (con't)

Legend
d = "don't care" condition

Event: NET_DELIVER(SOURCE ADDRESS, DESTINATION ADDRESS, PROTOCOL,
TOS[precedence, reliability, delay, throughput],
OPTIONS[security], LENGTH, DATA)

Actions:

Seq# status ?	RST on?	Sec prec match?	SYN in wndow	ACK status test2?	FIN ACK'd ?	
INVAL	NO	d	d	d	d	send_ack(sv.recv_next)
INVAL	YES	d	d	d	d	-- no action
VALID	NO	NO	d	d	d	reset(SEG);reset_self(SP) sv.state=CLOSED
VALID	NO	YES	NO	NONE	d	-- no action
VALID	NO	YES	NO	INVAL	d	send_ack(sv.recv_next)
VALID	NO	YES	NO	VALID	NO	update
VALID	NO	YES	NO	VALID	YES	start_time_wait;sv.state=TIME_WAIT
VALID	NO	YES	YES	d	d	reset(SEG);reset_self(SF) sv.state=CLOSED
VALID	YES	d	d	d	d	reset_self(RA); sv.state=CLOSED

```
==================
STATE = FIN_WAIT1
==================

Event: Allocate( LCN, DATA_LENGTH )

Actions: new_allocation


Event: Send()
       or Close()

Actions: error("Connection closing.")


Event: Active Open()
       or Active Open with data()
       or Full Passive Open()
       or Unspecified Passive Open()

Actions: error("Connection already exists.")


Event: Abort( LCN )

Actions: reset(CURRENT); reset_self(UA); sv.state=CLOSED


Event: Retransmission Timeout

Actions: retransmit


Event: ULP Timeout

Actions: reset(CURRENT); reset_self(UT); sv.state=CLOSED
```

STATE = FIN_WAIT1 (con't)

Legend
d = "don't care" condition

Event: NET_DELIVER(SOURCE ADDRESS, DESTINATION ADDRESS, PROTOCOL, TOS[precedence, reliability, delay, throughput], OPTIONS[security], LENGTH, DATA)

Actions:

Seq# status ?	RST on?	Sec prec match?	SYN in wndow	ACK status test2?	Zero recv wndow	FIN ACK'd ?	FIN on?	
INVAL	NO	d	d	d	d	d	d	send_ack(sv.recv_next)
INVAL	YES	d	d	d	d	d	d	-- no action
VALID	NO	NO	d	d	d	d	d	reset;reset_self(SP) sv.state=CLOSED
VALID	NO	YES	NO	NONE	d	d	d	-- no action
VALID	NO	YES	NO	INVAL	d	d	d	send_ack(sv.recv_next)
VALID	NO	YES	NO	VALID	NO	NO	NO	update;accept
VALID	NO	YES	NO	VALID	NO	NO	YES	update;accept;set_fin sv.state=CLOSING
VALID	NO	YES	NO	VALID	NO	YES	NO	update;accept sv.state=FIN_WAIT2
VALID	NO	YES	NO	VALID	NO	YES	YES	update;accept;set_fin start_time_wait sv.state=TIME_WAIT
VALID	NO	YES	NO	VALID	YES	NO	d	update
VALID	NO	YES	NO	VALID	YES	YES	d	update;sv.state=FIN_WAIT2
VALID	NO	YES	YES	d	d	d	d	reset(SEG);reset_self(SF) sv.state=CLOSED
VALID	YES	d	d	d	d	d	d	reset_self(RA); sv.state=CLOSED

```
==================
STATE = FIN_WAIT2
==================

Event: Abort( LCN )

Actions: reset(CURRENT); reset_self(UA); sv.state=CLOSED


Event: Allocate( LCN, DATA_LENGTH )

Actions: new_allocation


Event: Send()
       or Close()

Actions: error("Connection closing.")


Event: Active Open()
       or Active Open with data()
       or Full Passive Open()
       or Unspecified Passive Open()

Actions: error("Connection already exists.")


Event: Retransmission Timeout

Actions: retransmit


Event: ULP Timeout

Actions: reset(CURRENT); reset_self(UT); sv.state=CLOSED
```

STATE = FIN_WAIT2 (con't)

Legend
d = "don't care" condition

Event: NET_DELIVER(SOURCE ADDRESS, DESTINATION ADDRESS, PROTOCOL,
TOS[precedence, reliability, delay, throughput],
OPTIONS[security], LENGTH, DATA)

Actions:

Seq# status ?	RST on?	Sec prec match?	SYN in wndow	ACK status test2?	Zero recv wndow	FIN on?	
INVAL	NO	d	d	d	d	d	send_ack(sv.recv_next)
INVAL	YES	d	d	d	d	d	-- no action
VALID	NO	NO	d	d	d	d	reset(SEG);reset_self(SP) sv.state=CLOSED
VALID	NO	YES	NO	NONE	d	d	-- no action
VALID	NO	YES	NO	INVAL	d	d	send_ack(sv.recv_next)
VALID	NO	YES	NO	VALID	NO	NO	update;accept
VALID	NO	YES	NO	VALID	NO	YES	update;accept;set_fin start_time_wait sv.state=TIME_WAIT
VALID	NO	YES	NO	VALID	YES	d	update
VALID	NO	YES	YES	d	d	d	reset(SEG);reset_self(SF) sv.state=CLOSED
VALID	YES	YES	d	d	d	d	reset_self(RA);sv.state=CLOSED

```
================
STATE = LAST_ACK
================

Event: Abort( LCN )

Actions: reset_self(UA); sv.state=CLOSED

Event: Send()
       or Close()
       or Allocate()

Actions: error("Connection closing.")


Event: Active Open()
       or Active Open with data()
       or Full Passive Open()
       or Unspecified Passive Open()

Actions: error("Connection already exists.")


Event: Retansmission Timeout

Actions: retransmit


Event: ULP Timeout

Actions: reset(CURRENT); reset_self(UT); sv.state=CLOSED
```

STATE = LAST_ACK (con't)

Legend
d = "don't care" condition

Event: NET_DELIVER(SOURCE ADDRESS, DESTINATION ADDRESS, PROTOCOL,
TOS[precedence, reliability, delay, throughput],
OPTIONS[security], LENGTH, DATA)

Actions:

Seq# status ?	RST on?	Sec prec match?	SYN in wndow	ACK status test2?	FIN ACK'd ?	
INVAL	NO	d	d	d	d	send_ack(sv.recv_next)
INVAL	YES	d	d	d	d	-- no action
VALID	NO	NO	d	d	d	reset(SEG);reset_self(SP) sv.state=CLOSED
VALID	NO	YES	NO	NONE	d	-- no action
VALID	NO	YES	NO	INVAL	d	send_ack(sv.recv_next)
VALID	NO	YES	NO	VALID	NO	-- no action
VALID	NO	YES	NO	VALID	YES	reset_self(UC);sv.state=CLOSED
VALID	YES	YES	YES	d	d	reset(SEG);reset_self(SF) sv.state=CLOSED
VALID	YES	d	d	d	d	reset_self(RA);sv.state=CLOSED

```
================
STATE = TIME_WAIT
================

Event: Abort( LCN )

Actions: reset_self(UA); sv.state=CLOSED


Event: Send()
       or Close()
       or Allocate()

Actions: error("Connection closing.")


Event: Active Open()
       or Active Open with data()
       or Full Passive Open()
       or Unspecified Passive Open()

Actions: error("Connection already exists.")


Event: Time Wait Timeout

Actions: reset_self(UC); sv.state=CLOSED
```

STATE = TIME_WAIT (con't)

Legend
d = "don't care" condition

Event: NET_DELIVER(SOURCE ADDRESS, DESTINATION ADDRESS, PROTOCOL,
TOS[precedence, reliability, delay, throughput],
OPTIONS[security], LENGTH, DATA)

Actions:

Seq# status ?	RST on?	Sec prec match?	SYN in wndow	ACK status test2?	FIN on?	
INVAL	NO	d	d	d	d	send_ack(sv.recv_next)
INVAL	YES	d	d	d	d	-- no action
VALID	NO	NO	d	d	d	reset(SEG);reset_self(SP) sv.state=CLOSED
VALID	NO	YES	NO	NONE	d	-- no action
VALID	NO	YES	NO	INVAL	d	send_ack(sv.recv_next)
VALID	NO	YES	NO	VALID	NO	-- no action
VALID	NO	YES	NO	VALID	YES	send_ack(sv.recv_next) restart_time_wait
VALID	NO	YES	YES	d	d	reset(SEG);reset_self(SF) sv.state=CLOSED
VALID	YES	d	d	d	d	reset_self(RA);sv.state=CLOSED

6.3.6.2 Decision Functions The following functions examine information contained in interface parameters, interface data, and the state vector to make decisions. These decisions can be thought of as further refinements of the event and/or state. The return values of the functions represent decisions made.

6.3.6.2.1 ACK on? The ACK_on function determines whether the acknowledgement field of the incoming segment is in use.
The data effects of this function are:

- Data examined only: from_NET.seg.ack_flag

- Return values:
  NO -- indicates the ACK flag is false and the ACK number should not be examined

  YES -- indicates that the ACK flag is true and the ACK number is in use

```
if (from_NET.seg.ack_flag = TRUE )
then return (YES)
else return (NO);
```

6.3.6.2.2 ACK status test1? The ACK_status_test1 function compares the ACK number of the incoming segment with the current send variables to determine whether the ACK is valid. This function is intended for use during connection establishment when "old duplicate" ACKs cannot occur.
The data effects of this function are:

- Data examined only:
  from_NET.seg.ack_num sv.send_next
  from_NET.seg.ack_flag sv.send_una

- Return values:
  NONE -- no ACK appears in the incoming segment
  INVALID -- the incoming segment carries an ACK which is outside the send window
  VALID -- the incoming segment carries an ACK inside the send window which should be used for update

```
--During connection establishment, an ACK is valid if
--it falls inside the send window because old ACKs do not
--exist for this connection incarnation.

--Check for presence of ack flag.

    if (from_NET.seg.ack_flag = FALSE )
    then return (NONE)
    else  --Validate ACK against current send window

         if (from_NET.seg.ack_num =< sv.send_una)
             or (from_NET.seg.ack_num > sv.send_next)

         then return (INVALID )  --present but unacceptable

         else return (VALID);    --present and inside send window
```

6.3.6.2.3 ACK status test2? The ACK_status_test2 function examines the ACK number of the incoming segment against the current send variables to determine *whether the ACK is valid. This function is intended for use after connection* establishment when old duplicate ACKs can legally occur.
The data effects of this function are:

- Data examined only:
  from_NET.seg.ack_flag sv.send_next
  from_NET.seg.ack_num

- Return values:
  NONE -- no ACK appears in the incoming segment
  INVALID -- the incoming segment carries an ACK for something which has not yet been sent.
  VALID -- the incoming segment carries an ACK which either falls in the window (and should be used for update) or duplicates a previous ACK.

```
--After a connection is established, an ACK is valid if
--it ACKs something sent on this connection incarnation.

--Check for presence of ack flag.

    if (from_NET.seg.ack_flag = FALSE )
    then return (NONE)
    else  --Validate ACK against current send window

         if (from_NET.seg.ack_num > sv.send_next)

         then return (INVALID )  --present but unacceptable

         else return (VALID);    --present and okay
```

6.3.6.2.4 checksum check? The checksum_check function computes the checksum of an incoming segment and compares it against the checksum field in the header of the incoming segment.
The data effects of this function are:

- Data examined only:
  all fields of from_NET.seg
  from_NET.source_addr
  from_NET.destination_addr
  from_NET.protocol
  from_NET.length

- Return values:
  NO -- indicates that the computed checksum does not match the value in from_NET.seg.checksum

  YES -- indicates that the computed checksum matches the value in from_NET.seg.checksum

```
--The checksum algorithm is the 16 bit one's complement of the
--one's complement sum of all 16 bit words in the segment
--header and segment text.  If a segment contains an odd number
--of octets, the last octet is padded on the right with zeros
--to form a 16-bit word for checksum purposes.
--While computing the checksum, the checksum field itself is replaced
--with zeros.
--The checksum includes a 96-bit pseudo header prefixed to the
--actual TCP header.  The pseudo header contains the
--source address, the destination address, the protocol identifier
--and the length of the TCP segment (not counting the pseudo header)
--as passed by the NET_DELIVER service primitive.
--
--     +--------+--------+--------+--------+
--     |        Source Address              |
--     +-----------------------------------+
--     |     Destination Address           |
--     +-----------------------------------+
--     |  zero  |Protocol| Segment length  |
--     +-----------------------------------+
--
```

--The actual computation is implementation dependent.

6.3.6.2.5 FIN ACK'd? The FIN_ACK'd function examines the acknowledgement field of the incoming segment to determine whether this segment ACKs a previously sent FIN.
The data effects of this function are:

- Data examined only:
  from_NET.seg.ack_flag sv.send_finflag
  from_NET.seg.ack_num sv.send_next

- Return values:
  NO -- the incoming segment does not ACK the FIN
  YES -- the incoming segment does carry an ACK of the previously sent FIN

```
--The sv.send_finflag indicates that the ULP has
--issued a CLOSE. The FIN's sequence number is one less than
--sv.send_next.

  if (sv.send_finflag = TRUE )
  then
       if (from_NET.seg.ack_flag = TRUE) and
            from_NET.seg.ack_num = sv.send_next )
       then return (YES)
       else return (NO)
```

6.3.6.2.6 FIN on? The FIN_on function determines whether the incoming segment carries a FIN indicating the remote side has no more data to send.
The data effects of this function are:

- Data examined only: from_NET.seg.fin_flag

- Return values:
  NO -- the segment does not contain a FIN
  YES -- the segment does carry a FIN

```
--The segment header field seg.fin_flag indicates the
--presence or absence of a FIN.

  if (from_NET.seg.fin_flag = FALSE)
  then return (NO)
  else return (YES);
```

6.3.6.2.7 FIN seen? The FIN_seen function examines both the incoming segment and the sv.recv variables for the previous or current presence of a FIN from the remote TCP. This function is used in the established state because a FIN may have been recorded during connection opening.
The data effects of this function are:

- Data examined only:
  from_NET.seg.fin_flag sv.recv_finflag

- Return values:
  NO -- No FIN has been received from the remote TCP in this or previous segments.
  YES -- A FIN has been received, either in the incoming segment or a previous segment.

```
--A FIN received during connection opening is saved in
--sv.recv_finflag.  A FIN is present in an
--incoming segment if from_NET.seg.fin_flag is set true.

  if (( sv.recv_finflag = TRUE ) or
        (from_NET.seg.fin_flag = TRUE ))
  then return (YES)
  else return (NO);
```

6.3.6.2.8 open mode? The open_mode function determines what kind of open service request the local ULP issued.
The data effects of this function are:

- Data examined only: sv.open_mode

- Return values:
  ACTIVE -- the ULP requested an Active Open with or without data for this connection.
  PASSIVE -- the ULP request a Full Passive Open or an Unspecified Passive Open for this connection.

```
--The type of open request is recorded in sv.open_mode.

  if (sv.open_mode = PASSIVE)
  then return (PASSIVE)
  else return (ACTIVE);
```

6.3.6.2.9 sv prec vs seg prec? The sv_prec_vs_seg_prec function compares the precedence recorded in the state vector against the precedence level of the incoming segment.
The data effects of this function are:

- Data examined only:
  sv.original_prec from_NET.type_of_service.precedence

- Return values:
  LESS - precedence in sv < segment precedence
  EQUAL - precedence in sv = segment precedence
  GREATER - precedence in sv > segment precedence

```
if (sv.original_prec < from_NET.precedence )
then return (LESS)
else if (sv.original_prec = from_NET.precedence )
     then return (EQUAL)
     else return (GREATER);
```

6.3.6.2.10 resources suffic open? The resources_suffic_open function examines the internal resources available in this TCP entity to determine whether another connection can be supported.
The data effects of this function are:

- Data examined only: --implementation dependent

- Return values:
  NO -- indicates that another connection cannot be supported at this time.
  YES -- indicates that internal resources are sufficient to support another connection

```
--This function is based on the assumption that a TCP entity
--has finite resources made up of table space, storage capacity,
--and other implementation dependent areas.  Although the amount
--of these resources may either be fixed at system configuration
--or vary dynamically according to system usage, more connections
--may be requested than can be supported by the entity.

  if ( enough resources are available for another connection )
  then return (YES)
  else return (NO);
```

6.3.6.2.11 resources suffic send? The resources_suffic_send function examines the resources of this TCP connection to determine if more data can be accepted from the ULP for transfer.
The data effects of this function are:

- Data examined only: --implementation dependent

- Return values:
  NO -- indicates that data cannot be accepted from the ULP at this time.
  YES -- indicates that internal resources are sufficient to accept and transfer more ULP data

```
--This function is based on the assumption that a TCP
--connection has finite resources.  Although the amount of
--these resources may be fixed at connection establishment,
--or vary over connection lifetime, at some point a sending
--ULP might exceed the TCP entity's capacity.

  if ( enough resources are available to handle this SEND )
  then return (YES)
  else return (NO);
```

6.3.6.2.12 RST on? The RST_on function examines the reset flag of the incoming segment's header to determine the presence of a RST.
The data effects of this function are:

- Data examined only: from_NET.seg.rst_flag

- Return values:
  NO -- the incoming segment does not contain a RESET
  YES -- the incoming segment does carry a RESET

```
  if (from_NET.seg.rst_flag = FALSE)
  then return (NO)
  else return (YES);
```

6.3.6.2.13 sec match? The sec_match function compares the security parameters (including security level, compartment, transmission control code, and handling restrictions) defined in the state vector against those accompanying the incoming segment.
The data effects of this function are:

- Data examined only:
  from_NET.options[security] sv.sec

- Return values:
  NO -- The values in the state vector do not match those of the incoming segment.
  YES -- The security information exactly matches that in the state vector.

```
--The security information is not carried in the segment header
--but is passed by the network protocol entity in the
--NET_DELIVER option parameter.

  if (from_NET.options[security] = sv.sec)
  then return (YES)
  else return (NO);
```

6.3.6.2.14 sec prec allowed? The sec_prec_allowed function examines the security and precedence information requested by a ULP in a connection open request and based on the implementation environment (i.e. secure host, unclassified system, etc.) determines whether this TCP entity can support them.
The data effects of this function are:

- Data examined only:
  from_ULP.precedence from_ULP.security

- Return values:
  NO -- This TCP entity cannot support the requested security and precedence.
  YES -- The security and precedence requested can be supported.

--This decision is implementation dependent.

6.3.6.2.15 sec prec match? The sec_prec_match function compares the precedence level and security information (including security level, compartment, transmission control code, and handling restrictions) defined in the state vector against those of the incoming segment.
The data effects of this function are:

- Data examined only:
  from_NET.type_of_service.precedence sv.sec
  from_NET.options[security] sv.actual_prec

- Return values:
  NO -- The security and precedence of the segment do not match those of the state vector.
  YES -- The security and precedence DO match.

```
if ((sv.sec = from_NET.options[security]) and
    (sv.actual_prec = from_NET.type_of_service.precedence))

then return (YES)
else return (NO);
```

6.3.6.2.16 seq# status? The seq#_status function compares the sequence number of the incoming segment against the current recv variables in the state vector to determine whether the segment contains data in the recv window. The data effects of this function are:

- Data examined only: from_NET.seg.seq_num sv.recv_wndw
  sv.recv_next

- Return values:
  VALID -- This segment does not contain data within the recv window.
  INVALID -- This segment DOES contain data in the recv window.

```
--Due to zero length recv window and zero length segments,
--this decision function must examine four cases.
--These cases are expressed in the following conditional statements.

if (from_NET.length = 0)
then if (sv.recv_wndw = 0)
     then
          --When the segment contains no data, and the receive
          --window is closed, the segment sequence number
          --must equal the next expected to be acceptable.
          if (from_net.seg.seq_num = sv.recv_next)
          then return (YES)
          else return (NO)
     else
         --When the segment contains no data and the receive
         --window is open, the segment sequence number must
         --fall within the receive window.
         if (sv.recv_next =< from_NET.seg.seq_num) and
            (from_NET.seg.seq_num < sv.recv_next+sv.recv_wndw))
         then return (YES)
         else return (NO)

else if (sv.recv_wndw = 0)
     then
          --When the segment carries data and the receive
          --window is closed, although no data can be
          --accepted, the control information is acceptable
          --if the segment sequence number exactly matches
          --the next expected.
          if (from_net.seg.seq_num = sv.recv_next)
          then return (YES)
          else return (NO)
     else
         --When the segment carries data and the receive window
         --is open, the segment is acceptable if any data
         --falls within the receive window.
         if
           --Does the front of the data lie within the window?
          (((sv.recv_next =< from_NET.seg.seq_num)
            and (from_NET.seg.seq_num < sv.recv_next+sv.recv_wndw))
```

```
or
--Does the back of the data lie within the window?
((sv.recv_next =< from_NET.seg.seq_num+from_NET.length)
and (from_NET.length+from_NET.seg.seq_num <
                              sv.recv_next+sv.recv_wndw))
or
--Does the middle of the data lie within the window?
((sv.recv_next > from_net.seg.seq_num)
and ( sv.recv_next+sv.recv_wndw <
      (from_NET.length+from_NET.seg.seq_num))))
then return (YES)
else return (NO);
```

6.3.6.2.17 SYN on? The SYN_on function examines the SYN flag of the incoming segment. The data effects of this function are:

- Data examined only: from_NET.seg.syn_flag

- Return values:
  NO -- No SYN is present in the incoming segment.
  YES -- A SYN is present in the segment.

```
if (from_NET.seg.syn_flag = TRUE)
then return (YES)
else return (NO);
```

6.3.6.2.18 SYN in wndow? The SYN_in_wndow function determines whether an incoming segment contains a SYN, and if so, whether its sequence number lies in the recv window.
The data effects of this function are:

- Data examined only:
  from_NET.seg.syn_flag sv.recv_next
  from_NET.seg.seq_num sv.recv_wndw

- Return values:
  NO -- No SYN is present, or a SYN is present but it does not fall in the recv window.
  YES -- A SYN is present and falls in the recv window.

```
--After a connection is established, no segments should contain
--SYNs.  However, certain situations may produce a SYN.
--Shortly after a connection opens, a duplicate of the original
--SYN may arrive.  It will not lie in the recv window, having
--already been accepted.  Or, during a connection of long
--duration, very very rare error conditions may produce a SYN within
--the recv window.  This situation must be detected.

  if ((from_NET.seg.syn_flag = TRUE ) and
      (from_NET.seg.seq_num >= sv.recv_next) and
      (from_NET.seg.seq_num < sv.recv_next + sv.recv_wndw))

  then return (YES)
  else return (NO);
```

6.3.6.2.19 zero recv wndow? The zero_recv_wndow function examines the recv_variables to determine whether the recv window is zero, preventing the acceptance of any data from the remote TCP.
The data effects of this function are:

- Data examined only: sv.recv_wndw

- Return values:
  NO -- The recv window is not zero. Data can be accepted.
  YES -- The recv window IS zero. No data can be accepted.

```
  if (sv.recv_wndw = 0)
  then return (YES)
  else return (NO);
```

6.3.6.3 Action Procedures The following action procedures represent the set of actions performed by a TCP entity state machine. They are called by the state and event correspondence defined in Section 6.3.6. These procedures have been organized and designed for clarity and are provided as guidelines. Although implementors can reorganize for better performance, the data effects of the resulting implementations must not differ from those specified below.

Certain aspects of the actions described in the following procedures are subject to design choices. Specifically, the selection of strategies for handling retransmissions, sending acknowledgements, segmenting data, accepting data from the remote TCP, and delivering data to the ULP are governed by implementation dependent criteria. These strategies are encapsulated in "policy" procedures such as accept_policy. A policy procedure discusses the available approaches and returns information to an action procedure indicating appropriate processing. The policy procedures defined in the following section are: accept_policy, ack_policy, deliver_policy, retransmit_policy, and send_policy.

This specification is intended to be as detailed and accurate as possible without implying a particular implementation approach or environment. However, a difficulty lies in the manipulation of internal data storage areas which is, by nature, implementation dependent. Thus, a set of data management routines are defined to manipulate the queues for send and receive data while specific data structures (such as arrays, linked lists, or circular buffers) remain undefined. The state vector can record the send and receive variables in terms of sequence numbers because the data routines correlate sequence numbers to the physical position of data within the data structures. The data management routines defined in the following section are: dm_add_to_recv, dm_add_to_send, dm_copy_from_send, dm_remove_from_send, and dm_remove_from_recv.

The actions procedures invoke the execution environment primitives, defined in Section 7, to pass messages between protocol levels (TRANSFER), to read current time (CURRENT_TIME), and to set and cancel timers (SET_TIMER, CANCEL_TIMER).

6.3.6.3.1 accept The accept action procedure accepts data from the incoming segment and places it in the receive queue. The amount of data accepted is governed by the implementation dependent acceptance policy. An ACK is generated for the accepted data according to the ack policy which is implementation dependent. Also, some data may be delivered according implementation dependent delivery policies.
The data effects of this procedure are:

- Data examined:
  all fields in from_NET sv.recv_alloc

- Data modified:
  all fields of to_NET sv.recv_wndw
  sv.recv_next sv.recv_push
  sv.recv_urg

- Local variables: start_seq amount offset

```
--The accept_policy procedure returns how much
--data is to be accepted, its beginning sequence number,
--and its location within the incoming segment.

  accept_policy( amount, start_seq, offset );

  if (amount > 0)
  then
       dm_add_to_recv( start_seq, amount, offset );

     --Update the recv_next sequence number if necessary.
       if (sv.recv_next = start_seq)
       then sv.recv_next := start_seq + amount;

     --Record PUSH and URGENT information.
       if ((from_NET.seg.push_flag = TRUE) and
           (sv.recv_push < start_seq + amount))
       then sv.recv_push := start_seq + amount;

       if ((from_NET.seg.urg_flag = TRUE) and
           (sv.recv_urg < from_NET.seg.seq_num + from_NET.seg.urgptr))
       then sv.recv_urg := from_NET.seg.seq_num + from_NET.seg.urgptr;

--Refer to ack_policy to determine whether an ACK should be generated
--at this point.
  to_ack := ack_policy();
  if (to_ack = TRUE)
  then send_ack( sv.recv_next );

--If the allocation allows, deliver data to the ULP.

  if (sv.recv_alloc > 0)
  then deliver;
end;
```

6.3.6.3.2 accept policy As one of the policy procedures, accept_policy discusses the alternative strategies for accepting the data of incoming segments and returns to the calling procedure the number of data octets to be accepted.
The parameters are:

1. starting_seq - sequence number of the first octet of data to be accepted

2. quantity - the number of octets of data to be accepted

3. segment_data_offset - the position of the first data octet within the incoming segment's text to be accepted.

A TCP implementation may use one of several strategies to accept data within the receive window from an incoming segment.

1. Accept in-order data only. The acceptance test is:
   from_NET.seg.seq_num = sv.recv_next
   That is, the sequence number of the incoming segment must exactly equal the next sequence number expected to be received.

2. Accept any data within the receive window. The acceptance test has several parts:
   sv.recv_next =< from_NET.seg.seq_num
   =< sv.recv_next + sv.recv_wndw

   - or -

   sv.recv_next =< from_NET.seg.seq_num + length
   =< sv.recv_next + sv.recv_wndw

   That is, any portion of the text falling within the receive window (i.e. in the interval between the next sequence number expected to be received and the last sequence number in the window) is accepted.

The "in-order" strategy allows a simple acceptance test and a straightforward scheme for data storage. However, the loss of a single segment can result in the remote TCP retransmitting every succeeding segment. The "in-the-window" strategy requires a more involved acceptance test and a sophisticated data storage scheme to keep track of data accepted out of order. Also, as each segment is accepted, the data storage must be checked so that a contiguous interval of out-of-order data can be recognized. This strategy allows the remote TCP to retransmit only lost segments.

6.3.6.3.3 ack policy As one of the policy procedures, ack_policy discusses the alternative strategies for acknowledging data accepted from incoming segments and returns to the calling action procedure a boolean value indicating whether an acknowledgement should be sent.

A TCP implementation may apply one of two acknowledgement timing schemes.

a. When data is accepted from remote TCP, immediately generate an empty segment containing current acknowledgement information and return it to the remote TCP.

b. When data is accepted from remote TCP, record the need to acknowledge data in the state vector, but wait for an outbound segment with data on which to piggyback the ACK. However to avoid a long delay, set an "ack timer" to limit the delay to a reasonable interval. Thus, if no outbound segment with data is produced within the chosen ack timeout interval, the timer expires and an empty ACK segment is generated and sent to the remote TCP. If a data segment is produced before the timer expires, the timer is cancelled and the need to acknowledge record is erased from the state vector. (Note that no "ack timeout" event appears in this specification. This event and the resulting call to the send_ack action procedure should be added if the this approach is taken.)

The trade-off between the two approaches is processing time versus control overhead. The "automatic" ack approach is simple, but results in extra segment generation. The "timed" ack approach requires more processing but will reduce the number of segments generated on connections with two-way data transfer.

6.3.6.3.4 check urg The check_urg action procedure examines the header of the incoming segment to determine whether new urgent information is present. If so, the urgent pointer is recorded in the recv variables.
The data effects of this procedure are:

```
     Data examined:
          from_NET.seg.urg_flag     from_NET.seg.data_offset
          from_NET.seg.urgptr       from_NET.length
          from_NET.seg.seq_nu.:

   - Data modified:  sv.recv_urg

begin
 --Check urgent flag and urgent pointer.
   if (from_NET.seg.urg_flag = TRUE)

   then --Check to see if a new urgent pointer is present.
        if (sv.recv_urg < from_NET.seg.seq_num + from_NET.seg.urgptr)
        then
             if (sv.recv_urg < sv.recv_save)
             then --If the ULP is not in "urgent mode", it must be
                  --informed of the presence of urgent information.
                      --implementation dependent action
             sv.recv_urg := from_NET.seg.seq_num + from_NET.seg.urgptr;
end;
```

6.3.6.3.5 compute checksum The compute_checksum procedure computes the checksum of an outbound segment and places the value in the header's checksum field.

The data effects of this function are:

- Data examined:
  all fields of to_NET.seg to_NET.protocol
  to_NET.source_addr to_NET.length
  to_NET.destination_addr

- Data modified: to_NET.seg.checksum

```
begin
   --The checksum algorithm is the 16 bit one's complement of the
   --one's complement sum of all 16 bit words in the segment
   --header and segment text.  If a segment contains an odd number
   --of octets, the last octet is padded on the right with zeros
   --to form a 16-bit word for checksum purposes.  While computing
   --the checksum, the checksum field itself is replaced with zeros.
   --The checksum includes a 96-bit pseudo header prefixed to the
   --actual TCP header.  As, the diagrams shows, the pseudo header
   --contains the source address, the destination address, the
   --protocol identifier, and the length of the TCP segment
   --(not counting the pseudo header).
   --
   --      +--------+--------+--------+--------+
   --      |        Source Address              |
   --      +-----------------------------------+
   --      |     Destination Address           |
   --      +-----------------------------------+
   --      |  zero  |Protocol| Segment length  |
   --      +-----------------------------------+
   --
   --The actual computation is implementation dependent.
end;
```

6.3.6.3.6 conn_open The conn_open action procedure is called just before a connection enters the ESTABLISHED state. According to the implementation policy, the procedure updates the ACK and window information, delivers any waiting data, and acknowledges any received data. The ULP is notified of the newly opened connection with an OPEN_SUCCESS service response.
The data effects of the procedure are:

- Data examined: sv.lcn

- Data modified: to_ULP.service_response to_ULP.lcn

- Local variables: delivery_amount need_to_ack

```
begin
  --Inform the ULP.
       to_ULP.service_response := OPEN_SUCCESS;
       to_ULP.lcn := sv.lcn;
       TRANSFER to_ULP to the ULP identified by sv.source_port;

  --The incoming segment contained either a SYN and an ACK of
  --our SYN, or just an ACK.  In either case, use the new
  --ack to update the send variables.
       update;

  --Based on implementation dependent policies, deliver any waiting
  --data to the ULP.
       delivery_amount := deliver_policy();
       if (delivery_amount > 0) then deliver;

  --Based on implementation dependent acking policy, ack the
  --incoming segment.
       need_to_ack := ack_policy();
       if (need_to_ack = TRUE) then send_ack;
end;
```

6.3.6.3.7 deliver The deliver action procedure moves data accepted from the remote TCP into the to_ULP structure for delivery to the ULP. The amount of data delivered is based on the receive allocation, the amount of pushed data, and the implementation dependent delivery policy.
The data effects of this procedure are:

- Data examined:

sv.recv_push	sv.recv_next
sv.recv_urg	sv.recv_finflag
sv.lcn	sv.source_port

- Data modified:

sv.recv_save	all fields of to_ULP
sv.recv_alloc	

- Local variables: pushed delivery_amount urgent_length

```
begin
  --Does pushed data await delivery?
  if (sv.recv_push > sv.recv_save)
  then --Pushed data waits so compute amount needing delivery.
       if (sv.recv_push > sv.recv_next)
       then pushed := sv.recv_next - sv.recv_save
       else pushed := sv.recv_push - sv.recv_save;

       --Is there enough allocation for all the pushed data?
       if (sv.recv_alloc < pushed)
       then delivery_amount := sv.recv_alloc
       else delivery_amount := pushed;

  else --No pushed data waits. Refer to the deliver_policy
       --to determine how much data should be passed to the
       --ULP at this point.
         delivery_amount := deliver_policy();

  --Deliver computed amount of data to ULP, including urgent
  --information.

  if (delivery_amount > 0)
  then begin
    --Check for "end of urgent" in data which cannot be delivered
    --in the same delivery unit with subsequent non-urgent data.

      if ((sv.recv_urg > sv.recv_save) and
              (sv.recv_urg < sv.recv_save + delivery_amount))
      then --Deliver the urgent data alone first.
           begin
           urgent_length := sv.recv_urg - sv.recv_save;

           dm_remove_from_recv(sv.recv_save, urgent_length);
           to_ULP.data_length := urgent_length;
           --Note that implementation dependent delivery unit
```

```
            --size restrictions are not handled.
            to_ULP.urgent_flag := TRUE;
            to_ULP.lcn := sv.lcn;
            to_ULP.service_response := DELIVER;
            TRANSFER to_ULP to the ULP named by to_ULP.source_port;

            sv.recv_save := sv.recv_urg;
            sv.recv_alloc := sv.recv_alloc - urgent_length;
            delivery_amount := delivery_amount - urgent_length;
            end;

        --Move data without an end of urgent into to_ULP.data
        --and deliver to ULP.
          dm_remove_from_recv(sv.recv_save, delivery_amount);
          to_ULP.data_length := delivery_amount;
          --Note that implementation dependent delivery unit
          --size restrictions are not handled.
          to_ULP.lcn := sv.lcn;
          if (sv.recv_save < sv.recv_urg)
          then to_ULP.urgent_flag := TRUE
          else to_ULP.urgent_flag := FALSE;

          TRANSFER to_ULP to the ULP named by sv.source_port;

        --Update recv variables.
          sv.recv_save := sv.recv_save + delivery_amount;
          sv.recv_alloc := sv.recv_alloc - delivery_amount;

        --If the remote side has closed, and this data clears the
        --receive queue, the ULP must be notified.
          if ((sv.recv_finflag = TRUE) and
              (sv.recv_next = sv.recv_save))
          then
              begin
              to_ULP.service_response := CLOSING;
              to_ULP.lcn := sv.lcn;
              TRANSFER to_ULP to the ULP named by sv.source_port;
              end;

        end; --of data delivery to ULP
      end;
    end;
```

6.3.6.3.8 deliver policy As one of the policy procedures, deliver_policy discusses the alternative strategies for delivering data to the ULP. It returns to the calling procedure the number of octets of data to be delivered.

Barring zero receive allocations and pushed data, the TCP specification allows an implementation to deliver data to the ULP at its own convenience. However, performance considerations should be examined. On one hand, data available for delivery should be delivered with reasonable promptness; it should not be delayed indefinitely while waiting for a delivery units worth to arrive. On the other hand, the data should not be delivered an octet at a time wasting both internal TCP processing time and external execution environment resources. A reasonable compromise can be achieved guided by system design criteria.

6.3.6.3.9 dispatch The dispatch action procedure accepts the data and interface parameters passed by the ULP in a send request, adds the data to the send queue, and adjusts appropriate send variables. Depending on the send policy, the procedure may segment and transmit some portion of data to the remote TCP. The data effects of this procedure are:

- Data examined:
  - from_ULP.lcn
  - from_ULP.data
  - from_ULP.data_length
  - from_ULP.push_flag
  - from_ULP.urgent_flag
  - from_ULP.ulp_timeout
- Data modified:
  - sv.send_free
  - sv.ulp_timeout
  - sv.send_push
  - sv.send_urg
  - sv.send_next
  - sv.send_una
  - sv.send_wndw

```
begin
   --Save the data along with timestamp, starting at sv.send_free,
   --then update the send variables.

     add_to_send(sv.send_free, from_ULP.data_length, CURRENT_TIME());

     sv.send_free := sv.send_free + from_ULP.length;

     if (from_ULP.push_flag = TRUE)
     then sv.send_push := sv.send_free;

     if (from_ULP.urgent_flag = TRUE)
     then sv.send_urg := sv.send_free;

   --Depending on implementation, the ULP timeout timer may
   --need to be restarted when the interval is changed by the ULP.
     if ((from_ULP.ulp_timeout !!= NULL)   --option exercised to set timeout
        and (from_ULP.ulp_timeout !!= sv.ulp_timeout))
     then sv.ulp_timeout := from_ULP.ulp_timeout;

   --Call the send_new_data procedure to determine if any
   --newly received data can be sent at this time.

      send_new_data;

     end; --non-zero send window processing
end;
```

6.3.6.3.10 dm add to send As one of the data management routines, the dm_add_to_send procedure adds the data provided by the ULP in from_ULP.data to the send storage area.
The calling sequence is :

dm_add_to_send(seq_num, length, time)

seq_num = the sequence number of the first octet being added to the storage area

length = the number of octets to be added

time = the current time to be associated with each data octet for later determination of data age for the ULP timeout.

6.3.6.3.11 dm add to recv As one of the data management routines, the dm_add_to_recv procedure copies data from an incoming segment, found in from_NET.seg.data, into the receive storage area. This routine is called as segments are validated and portions of their data are found to be in the receive window.
The calling sequence is:

dm_add_to_recv(seq_num, length, offset)

seq_num = the sequence number of the first octet to be copied.

length = the number of data octet to be copied.

offset = the location of first octet to be taken from the data portion of the segment.

6.3.6.3.12 dm copy from send As one of the data management routines, the dm_copy_from_send procedure copies data from the send storage area into to_NET.seg.data. This routine is used as data is segmented and transmitted initially, and as retransmissions are required.
The calling sequence is:

dm_copy_from_send(seq_num, length)

seq_num = the sequence number of the first data octet to be copied into to_NET.seg.data

length = the number of octets to be copied

6.3.6.3.13 dm remove from recv As one of the data management routines, the dm_remove_from_recv routine removes data from the receive storage area and places it in the to_ULP.data structure. This is called as data is delivered to the ULP.
The calling sequence is:

```
dm_remove_from_recv( seq_num, length )

   seq_num = the sequence number of the first octet to
             be removed and copied

   length = the number of data octets to be removed and copied
```

6.3.6.3.14 dm remove from send As one of the data management routines, the dm_remove_from_send procedure deletes data from the send storage area. This routine is called as data is acknowledged by the remote TCP and removed from the retransmission "queue."
The calling sequence is:

```
dm_remove_from_send( seq_num, length )

   seq_num = the sequence number of the first octet to be removed.

   length = the number of data octets to be removed.
```

6.3.6.3.15 error The error procedure fills in the fields of to_ULP with the local connection name, the ERROR service response, and the error description passed by parameter. This information is passed to the local ULP.

```
   - Data examined: sv.lcn              sv.source_port

   - Data modified:
                    to_ULP.lcn          to_ULP.service_response
                    to_ULP.error_desc

begin
     --Construct an error message for the local ULP.

     to_ULP.service_response := ERROR;
     to_ULP.lcn := sv.lcn;
     to_ULP.error_desc := parameter;

     TRANSFER to_ULP to the ULP named by sv.source_port;
end;
```

6.3.6.3.16 format net params The format_net_params procedure fills in the parameters used by the network protocol entity after the calling procedure has filled in the outgoing segment header. The size of the segment's text portion is passed by parameter.

- Data examined:
  - to_NET.seg.data_offset
  - sv.destination_addr
  - sv.source_addr
  - sv.destination_port
  - sv.source_port

- Data modified:
  - to_NET.identifier
  - to_NET.type_of_service
  - to_NET.protocol
  - to_NET.length
  - to_NET.destination_addr
  - to_NET.source_addr
  - to_NET.seg.source_port
  - to_NET.destination_port
  - to_NET.dont_fragment

```
begin
   --Fill in the network parameters.
        to_NET.seg.source_port      := sv.source_port;
        to_NET.seg.destination_port := sv.destination_port;
        to_NET.source_addr          := sv.source_addr;
        to_NET.destination_addr     := sv.destination_addr;
        to_NET.protocol             := TCP_ID;
        to_NET.type_of_service.precedence  := sv.actual_prec;
        to_NET.type_of_service.reliability := NORMAL;
        to_NET.type_of_service.delay       := NORMAL;
        to_NET.type_of_service.throughput  := NORMAL;
        to_NET.identifier                  := gen_id();
        to_NET.dont_fragment        := FALSE;
        to_NET.time_to_live         := ONE_MINUTE_TTL;
        to_NET.length               := to_NET.seg.data_offset + parameter;
        to_NET.options[security]    := sv.sec;
end;
```

6.3.6.3.17 gen_id The gen_id action procedure returns an identifier to the calling procedure to be passed to the network protocol entity when a segment is transmitted with a NET_SEND primitive.

- Data examined: --implementation dependent

- Data modified: --none

```
begin
   --The generation of the identifier is implementation dependent.
   --The network protocol entity uses the identifier, along with
   --addressing information, to distinguish between sending units
   --(i.e. datagrams) if fragmentation and reassembly are required.
   --So, TCP must generate unique identifiers for each segment if
   --the data is to be transmitted without confusion.
   --
   --Also, if a retransmitted segment is accompanied by the
   --identifier used for its original transmission, the network
   --protocol entity may be able piece together parts of the
   --original and the retransmission to improve its performance.
   --Note that if repackaging is performed during retransmission,
   --the original identifier cannot be used.

end;
```

6.3.6.3.18 gen_isn The gen_isn procedure returns an initial sequence number to the calling procedure for use during the three-way handshake of connection establishment.

- Data examined: --implementation dependent

- Data modified: - none -

  --implementation dependent action

6.3.6.3.19 gen_lcn The gen_lcn procedure returns a local connection name, or lcn, to the calling procedure to be used as a shorthand identifier by TCP and the local ULP in service requests and responses pertaining to a connection.

- Data examined: --implementation dependent

- Data modified: - none -

```
begin
     --The generation of the lcn is implementation dependent.
     --A TCP entity usually supports many connections.
     --If the lcn is a pointer or table index, service requests
     --can be quickly matched to their state vector.
end;
```

6.3.6.3.20 gen_syn The gen_syn action procedure formats and transmits a segment containing a SYN to the remote TCP. As part of the SYN generation, an initial sequence number is selected.

The procedure accepts one parameter whose values are ALONE, WITH_ACK, and WITH_DATA, indicating whether the segment will contain an ACK or data. This procedure does not handle generating a SYN carrying a FIN flag because the specified service interface does support a transaction primitive described in Appendix C. However, if such primitive were created, this procedure would have to be modified to handle it.
The data effects of this procedure are:

- Data examined:

sv.source_port	sv.recv_isn
sv.source_addr	sv.recv_next
sv.destination_port	sv.send_next
sv.destination_addr	sv.send_free
sv.recv_wndw	sv.send_push
sv.send_urg	

- Data modified:

sv.send_isn	sv.send_next
all fields of to_NET	sv.send_una

- Local variables: amount

```
begin
     --Generate the initial sequence number to be used for
     --data sent to the remote TCP.
     sv.send_isn := gen_isn();
     sv.send_next := sv.send_isn +1;  --SYN uses the first seq#.
     sv.send_una := sv.send_isn;

     to_NET.seg.seq_num := sv.send_next;

     --Check parameter to determine exact type of SYN.
     case parameter of

          when ALONE =>
               to_NET.seg.ack_flag := FALSE;
               to_NET.seg.wndw     := 0;
               to_NET.seg.push_flag:= FALSE;
               to_NET.seg.urg_flag := FALSE;
               amount := 0;

          when WITH_ACK =>
               to_NET.seg.ack_flag := TRUE;
               to_NET.seg.wndw     := sv.recv_wndw;
               to_NET.seg.ack_num := sv.recv_isn + 1;
               to_NET.seg.push_flag := FALSE;
               to_NET.seg.urg_flag := FALSE;
               amount := 0;
```

```
          when WITH_DATA =>
                 to_NET.seg.ack_flag := FALSE;
                 to_NET.seg.wndw     := 0;

               --The data supplied by the ULP is in the send queue.
               --However, the amount of data to accompany the SYN
               --is determined by the send_policy.

                 amount := send_policy();

                 if (amount > 0)
                 then dm_copy_from_send( sv.send_next, amount );
                      if (sv.send_push = sv.send_next + amount)
                      then to_NET.seg.push_flag := TRUE;
                      sv.send_next := sv.send_next + amount;
                 else to_NET.seg.push_flag := FALSE;
          end case;

     --Add the urgent information regardless of data length.
       if (sv.send_urg >= to_NET.seg.seq_num)
       then to_NET.seg.urg_flag := TRUE;
            to_NET.seg.urgptr := sv.send_urg - to_NET.seg.seq_num;
       else to_NET.seg.urg_flag := FALSE;

     if (MAX_SEGMENT_SIZE option used in this implementation)
     then
          to_NET.seg.options[1]    := 2;    --Max header size option kind
          to_NET.seg.options[2]    := 4;    --option length = 4 octets
          to_NET.seg.options[3..4] := MAX_SEGMENT_SIZE;  --impl.dep value
          to_NET.seg.data_offset   := 6;
     else
          to_NET.seg.data_offset  := OPTIONLESS_HEADER;

     format_net_params(amount);
     compute_checksum;
     TRANSFER to_NET to the network protocol entity.
end;
```

6.3.6.3.21 new allocation The new_allocation action procedure takes the new value provided by the ULP in an allocation service request and adds it to the current receive allocation. Data waiting for this allocation is delivered to the ULP.
The data effects of this procedure are:

- Data examined: from_ULP.data_length

- Data modified: sv.recv_alloc

```
begin
  --Add in the new receive allocation.
     sv.recv_alloc := sv.recv_alloc + from_ULP.data_length;

  --Depending on implementation dependent window management strategy,
  --this new receive allocation may be factored into a new
  --value for the receive window.

  --If data awaits this allocation, deliver it.
     deliver;
end;
```

6.3.6.3.22 open The open action procedure records the parameters from an open service request (either Active Open, Fully Specified Passive Open, or Unspecified Passive Open), assigns a local connection name, and returns it to the ULP in an OPEN_ID service response.
The data effects of this procedure are:

- Data examined:

from_ULP.request_name	
from_ULP.source_port	from_ULP.precedence
from_ULP.destination_port	from_ULP.security
from_ULP.destination_addr	from_ULP.timeout

- Data modified:

sv.source_port	sv.lcn
sv.source_addr	sv.original_prec
sv.destination_port	sv.sec
sv.destination_addr	sv.ulp_timeout
to_ULP.service_response	to_ULP.destination_addr
to_ULP.source_port	to_ULP.destination_port
to_ULP.source_addr	to_ULP.lcn

```
begin
     --Assign a local connection name according to
     --implementation dependent algorithms.
          sv.lcn := gen_lcn();

     --The security, precedence, and timeout parameters are
     --optional.  If they are not provided by the ULP, default
     --values are assigned.  For security and precedence defaults
     --in nonsecure environments, the lowest levels are generally used.
     --A timeout default is more arbitrary, but the current
     --suggested value is two minutes.

          if (from_ULP.security is present)
          then sv.sec := from_ULP.security
          else sv.sec := DEFAULT_SECURITY;

          if (from_ULP.precedence is present)
          then sv.original_prec := from_ULP.precedence;
               sv.actual_prec   := from_ULP.precedence;
          else sv.original_prec := DEFAULT_PRECEDENCE;
               sv.actual_prec   := DEFAULT_PRECEDENCE;

          if (from_ULP.timeout is present)
          then sv.ulp_timeout := from_ULP.timeout
          else sv.ulp_timeout := DEFAULT_TIMEOUT;


     --The source port is provided in all open requests. The source
     --address is the address of this TCP entity.
          sv.source_port := from_ULP.source_port;
          sv.source_addr := THIS_ADDRESS;
```

```
     --The remaining parameters vary according to open request type.

          case from_ULP.request_name of

               when Unspecified_Passive_Open =>
                    --This request does not carry the destination
                    --socket.  It remains unassigned until a matching
                    --SYN from a remote TCP arrives.
                    sv.open_mode := PASSIVE;

               when Full_Passive_Open =>
                    sv.destination_addr := from_ULP.destination_addr;
                    sv.destination_port := from_ULP.destination_port;
                    sv.open_mode := PASSIVE;

               when Active_Open =>
                    sv.destination_addr := from_ULP.destination_addr;
                    sv.destination_port := from_ULP.destination_port;
                    sv.open_mode := ACTIVE;

               when Active_Open_With_Data =>
                    sv.destination_addr := from_ULP.destination_addr;
                    sv.destination_port := from_ULP.destination_port;
                    sv.open_mode := ACTIVE;

                  --Record data accompanying open request.
                    save_send_data;
          end case;

     --Return the local connection name assigned.
          to_ULP.service_response := OPEN_ID;
          to_ULP.source_port := sv.source_port;
          to_ULP.source_addr := sv.source_addr;
          to_ULP.destination_addr := sv.destination_port;
          to_ULP.destination_port := sv.destination_addr;
          to_ULP.lcn := sv.lcn;

          TRANSFER to_ULP to the ULP named by sv.source_port;
end;
```

6.3.6.3.23 openfail The openfail action procedure informs the ULP that the attempted connection could not be opened. It also clears the state vector. The data effects of the procedure are:

- Data examined: sv.lcn sv.source_port

- Data modified:
  all state vector elements
  to_ULP.lcn to_ULP.service_response

```
--Construct an OPEN_FAIL message for the ULP.
     to_ULP.service_response := OPEN_FAIL;
     to_ULP.lcn := sv.lcn;
     TRANSFER to_ULP to the ULP named by sv.source_port;

--The state vector is cleared without generating a ULP message.
     reset_self(NO_REPORT);
```

6.3.6.3.24 part reset The part_reset action procedure clears the send and recv variables without terminating the connection.
The data effects of the procedure are:

- Data examined: sv.open_mode

- Data modified: all send and receive variables

```
begin
  --The remote TCP address and port are cleared if the connection
  --open mode was PASSIVE.
     if (sv.open_mode = PASSIVE)
     then
          sv.destination_port := NULL;
          sv.destination_addr := NULL;

  --Clear all variables set during the connection opening
  --handshake.
     dm_remove_from_send(sv.send_una, QUEUE_SIZE);
     dm_remove_from_recv(sv.recv_free, QUEUE_SIZE);

     sv.actual_prec   := NULL;     sv.send_next     := NULL;
     sv.recv_isn      := NULL;     sv.send_una      := NULL;
     sv.recv_next     := NULL;     sv.send_wndw     := NULL;
     sv.recv_wndw     := NULL;     sv.send_push     := NULL;
     sv.recv_alloc    := NULL;     sv.send_urg      := NULL;
     sv.recv_push     := NULL;     sv.send_finflag  := NULL;
     sv.recv_urg      := NULL;     sv.send_free     := NULL;
     sv.recv_save     := NULL;     sv.send_lastup1  := NULL;
     sv.recv_finflag  := NULL;     sv.send_lastup2  := NULL;
     sv.send_isn      := NULL;     sv.send_max_seg  := NULL;

end;
```

6.3.6.3.25 raise prec The raise precedence action procedure raises the precedence level recorded in the state vector to the level provided by the remote TCP. Section 6.1.9 of the entity overview discusses precedence negotiation during connection establishment.
The data effects of this procedure are:

- Data examined: from_NET.type_of_service.precedence

- Data modified: sv.actual_prec

```
  --A SYN from the remote TCP carries a precedence level
  --greater than that indicated by the local ULP.
  --Precedence is carried as a type of service parameter.
    sv.actual_prec := from_NET.type_of_service.precedence;
```

6.3.6.3.26 record_syn The record_syn action procedure records the control information from the incoming segment containing a SYN flag.
The data effects of this procedure are:

- Data examined: all fields of from_NET

- Data modified:

sv.recv_next	sv.send_wndw
sv.recv_urg	sv.send_una
sv.recv_isn	sv.destination_port
sv.send_max_seg	sv.destination_addr
sv.recv_push	

- Local variables: start_seq amount offset

```
begin
   --If this half of the connection was opened passively, the
   --remote information should be added to the state vector.
     if (sv.open_mode = PASSIVE)
     then
          sv.destination_port := from_NET.seg.source_port;
          sv.destination_addr := from_NET.source_addr;

   --Record recv_data.
     sv.recv_isn := from_NET.seg.seq_num;
     sv.recv_next := sv.recv_isn;

   --Record send data.
     if (from_NET.seg.ack_flag = TRUE)
     then sv.send_una := from_NET.seg.ack_num;

   --Record maximum segment size if present in option field.
     if ((from_NET.seg.data_offset > 5)     --optionless header size
         and (from_NET.seg.options[0] = 2)) --Max Seg Option Kind
     then
         sv.send_max_seg := from_NET.seg.option[3..4];

   --If data accompanied the SYN, apply the implementation
   --dependent data acceptance policy to determine how much
   --data should be saved, its position in the recv_queue,
   --and its position in the incoming segment.

        accept_policy( start_seq, amount, offset );

        if (amount > 0)
        then
             add_to_recv( start_seq, amount, offset );

           --Update the recv_next sequence number if necessary.
             if (sv.recv_next = start_seq)
             then sv.recv_next := start_seq + amount;
             else --record data position in receive storage area
```

```
                --implementation dependent action

            --Record PUSH and URGENT information.
              if ((from_NET.seg.push_flag = TRUE) and
                  (sv.recv_push < start_seq + amount))
              then sv.recv_push := start_seq + amount;

              if ((from_NET.seg.urg_flag = TRUE) and
                  (sv.recv_urg < from_NET.seg.seq_num + from_NET.seg.urgptr))
              then --record the new urgent data position
                  sv.recv_urg := from_NET.seg.seq_num + from_NET.seg.urgptr;
end;
```

6.3.6.3.27 reset The reset action procedure formats and send a segment with a reset flag to the remote TCP to terminate the connection. RESET segments must be formatted so that the remote TCP finds the segments acceptable. The procedure accepts one parameter indicating the format of the RESET segment to be sent.

The parameter value "SEG" indicates that the incoming segment determines the format. If the segment contains an ACK, this forms the basis of the sequence number in the RESET segment. If the segment does not contain an ACK, the RESET segment is made acceptable by carrying an ACK the incoming segment's text.

The parameter value CURRENT indicates that the RESET is not the result of an incoming segment, but because of a ULP abort request or the ULP timeout. In such situations, the RESET segment is formed with a sequence number based on current state vector values.
The data effects of this procedure are:

- Data examined:

sv.source_port	sv.sec
sv.source_addr	sv.actual_prec
sv.destination_port	sv.send_next
sv.destination_addr	sv.recv_next

- Data modified: -none-

```
begin
   --Based on the parameter, set the sequence and ack numbers.
     if (parameter = SEG)
     then --Check the incoming segment for ACK presence.
          if (from_NET.seg.ack_flag = TRUE)
          then
               to_NET.seg.seq_num     := from_NET.seg.ack_num;
               to_NET.seq.ack_flag    := FALSE;
          else
               to_NET.seg.seq_num := 0;
               to_NET.seg.ack_flag := TRUE;
               to_NET.seg.ack_num := from_NET.seg.seq_num +
                         (from_NET.length - from_NET.seg.data_offset*4);

      else --parameter = CURRENT, so use current state vector values.
           to_NET.seg.seq_num     := sv.send_next;
           to_NET.seq.ack_flag    := FALSE;

   --Form a segment using current state vector data, set the
   --reset flag, and transmit to the remote TCP.
        to_NET.seg.rst_flag    := TRUE;
        to_NET.seg.syn_flag    := FALSE;
        to_NET.seg.urg_flag    := FALSE;
        to_NET.seg.push_flag   := FALSE;
        to_NET.seg.fin_flag    := FALSE;
        to_NET.seg.window      := 0;
```

```
        to_NET.seg.data_offset := OPTIONLESS_HEADER;

        format_net_params( 0 );
        compute_checksum;
        TRANSFER to_NET to the network protocol entity;
end;
```

6.3.6.3.28 reset_self The reset_self action procedure informs the ULP that the connection is terminating, and then sets the state vector elements to their initial values.

The reset_self procedure has one parameter indicating the reason for connection termination. If the parameter equals NO_REPORT, no service response is prepared for the ULP. All other values produce service responses including RR for remote reset, NF for network failure, UT for ULP timeout, SP for security or precedence mismatch, UC for user close, and UA for user abort.
The data effects of this procedure are:

- Data examined: sv.lcn

- Data modified: all state vector elements

```
begin
if parameter != NO_REPORT
then begin
   case parameter of

      when RA =>
           to_ULP.error_desc := "Remote abort."
      when NF =>
           to_ULP.error_desc := "Network failure."
      when SP =>
           to_ULP.error_desc := "Security/precedence mismatch."
      when UT =>
           to_ULP.error_desc := "ULP timeout."
      when UA =>
           to_ULP.error_desc := "ULP abort."
      when UC =>
           to_ULP.error_desc := "ULP close."
      when SF =>
           to_ULP.error_desc := "Service failure."
   end case;

   to_ULP.lcn := sv.lcn;
   to_ULP.service_response := TERMINATE;
   TRANSFER to_ULP to the ULP identified by sv.source_port;
   end;

--Regardless of the cause, clear all queues and initialize state vector.

   part_reset;                          sv.lcn  := NULL;
   sv.source_port := NULL;              sv.sec  := NULL;
   sv.source_addr := NULL;              sv.original_prec := NULL;
   sv.destination_port := NULL;         sv.actual_prec   := NULL;
   sv.destination_addr := NULL;         sv.ulp_timeout   := NULL;
end;
```

6.3.6.3.29 restart time wait The restart_time_wait action procedure restarts the currently running "time wait" timer. This procedure is called after a retransmitted FIN is seen from the remote TCP.
The data effects of this procedure are:

- Data examined: - none -

- Data modified: - none -

```
--Cancel the existing timer and start it up from scratch.
   cancel_timer( TIME_WAIT, sv.lcn );
   start_timer( TIME_WAIT, sv.lcn, TIME_WAIT_INTERVAL );
```

6.3.6.3.30 retransmit The retransmit actions procedure resends data that has not been acknowledged within the retransmission timeout interval. Because the amount of data resent is implementation dependent, this decision is encapsulated in the retransmit_policy procedure.

The data effects of this procedure are:

Data examined:

sv.send_una	sv.send_wndw
sv.send_next	sv.send_max_seg
sv.send_push	sv.send_urg
sv.send_finflag	sv.send_free
sv.recv_next	sv.recv_wndw

Data modified:

all fields of to_NET
retransmission timer

- Local variables: retrans_amount start_pt pushed_amount

```
begin
 --Determine how much data should be retransmitted to the
 --remote TCP.
  retrans_amount := retransmit_policy();

  if (retrans_amount > 0)
  then
     begin
       --Starting from the front of the retransmission queue,
       --segment and retransmit data indicated by amount.

       start_pt := sv.send_una;
       to_NET.seg.seq_num     := start_pt;
       to_NET.seg.rst_flag    := FALSE;

       if (start_pt = sv.send_isn)
       then  --The SYN is being retransmitted.
          to_NET.seg.syn_flag    := TRUE;
          if (sv.recv_isn = NULL)  --Has the remote TCP been heard from?
          then to_NET.seg.ack_flag := FALSE;
               to_NET.seg.wndw        := 0;
          else to_NET.seg.ack_num     := sv.recv_next;
               to_NET.seg.ack_flag    := TRUE;
               to_NET.seg.wndw        := sv.recv_wndw;
          if (MAX_SEGMENT_SIZE option used in this implementation)
          then
               to_NET.seg.options[1]    := 2;  --See section 6.2.11
               to_NET.seg.options[2]    := 4;  --for option format.
               to_NET.seg.options[3..4] := MAX_SEGMENT_SIZE;
               to_NET.seg.data_offset   := 6;
          else to_NET.seg.data_offset   := OPTIONLESS_HEADER;

       else  --Normal data retransmission.
          to_NET.seg.ack_num     := sv.recv_next;
```

```
          to_NET.seg.ack_flag    := TRUE;
          to_NET.seg.syn_flag    := FALSE;
          to_NET.seg.data_offset := OPTIONLESS_HEADER;
          to_NET.seg.wndw        := sv.recv_wndw;

        --Note that this section assumes that this segment's size
        --is less than sv.send_max_seg.

      --The end of pushed data cannot be packaged with
      --subsequent non-pushed data.



    --Prepare and transmit data.

          dm_copy_from_send( sv.send_una, retrans_amount );

          --If pushed data within or following data in this segment,
          --set the PUSH flag to inform remote TCP.
          if (sv.send_una =< sv.send_push)
          then to_NET.seg.push_flag := TRUE
          else to_NET.seg.push_flag := FALSE;

          --If urgent data lies within or follows data in this segment,
          --record urgent data position in header.
          if (sv.send_urg > start_pt)
          then  to_NET.seg.urg_flag := TRUE;
                to_NET.seg.urgptr := sv.send_urg - start_pt;
          else  to_NET.seg.urg_flag := FALSE;

          --If this segment contains that last octet of data from
          --the ULP, set the FIN to inform the remote TCP.
          if ((sv.send_finflag = TRUE) and
              (sv.send_free = start_pt + retrans_amount))
          then to_NET.seg.fin_flag := TRUE
          else to_NET.seg.fin_flag := FALSE;

          format_net_params( retrans_amount );
          compute_checksum;
          TRANSFER to_NET to the network protocol entity;
          end; --of preparation and retransmission of unpushed data.
end;
```

6.3.6.3.31 retransmit policy As one of the policy procedures, retransmit_policy discusses the alternative strategies for retransmissions. It returns to the calling action procedure the number of octets to be retransmitted.

A TCP implementation may employ one of several retransmission strategies.

- First only retransmission - Maintain one retransmission timer for the entire queue. When the retransmission timer expires, send the segment at the front of the retransmission queue. Initialize the timer.

- Batch retransmission - Maintain one retransmission timer for the entire queue. When the retransmission timer expires, send all the segments on the retransmission queue. Initialize the timer.

- Individual retransmission - Maintain one timer for each segment on the retransmission queue. As the timers expire, the retransmit the segments individually and reset their timers.

The first only retransmission strategy is efficient in terms traffic generated because only lost segments are retransmitted; but the strategy can cause long delays. The batch retransmission creates more traffic but decreasing the likelihood of long delays. However, the actual effectiveness of either scheme depends in part on the acceptance policy of the receiving TCP.

For example, suppose a sending TCP sends three segments, all within the send window, to a receiving TCP. The first segment is lost by the network. A receiving TCP using the "in-order" acceptance strategy discards the second and third segments. A receiving TCP using the "in-window" strategy accepts the second and third segments, but does not acknowledge or deliver any data until the lost segment arrives.

Batch retransmission fits better with the in-order acceptance strategy because the receiving TCP has discarded all segments. All three segments must be retransmitted--the sooner the better. First-only retransmission fits better with the in-window acceptance policy because only the needed retransmission occurs because the receiving TCP has kept the segments within its receive window and awaits only the lost segment. The sending TCP may also choose to repackage segments for retransmission.

6.3.6.3.32 save fin The save_fin action procedure records the presence of a FIN flag in an incoming segment received before a connection is ESTABLISHED. The FIN is processed only in the ESTABLISHED state.
The data effects of the procedure are:

- Data examined: sv.recv_next

- Data modified: sv.recv_fin sv.recv_push

```
--Record FIN is recv_variable.
  sv.recv_finflag := TRUE;
  sv.recv_push := sv.recv_next;   --The PUSH function is assumed.
```

6.3.6.3.33 save send data The save_send_data action procedure saves the data provided by the local ULP in a "Send" or an "Active Open with Data" service request issued before the connection is ESTABLISHED.
The data effects of the procedure are:

- Data examined only:
  from_ULP.data from_ULP.length
  from_ULP.push_flag from_ULP.urgent_flag

- Data modified:
  sv.send_free sv.send_urg
  sv.send_push

```
begin
   --Take the data and add it to the send queue.
        dm_add_to_send( sv.send_free, from_ULP.length );
        sv.send_free := sv.send_free + from_ULP.length;

   --Set the urgent and push information as needed.
        if (from_ULP.push_flag = TRUE)
        then sv.send_push := sv.send_free;

        if (from_ULP.urg_flag = TRUE)
        then sv.send_urg := sv.send_free;
end;
```

6.3.6.3.34 send_ack The send_ack procedure formats and sends an empty segment with the ACK value indicated by parameter.
The data effects of this procedure are:

- Data examined:
  sv.send_next sv.source_port
  sv.recv_next sv.destination_port
  sv.actual_prec sv.sec

- Data modified: all to_NET.seg fields

```
begin
   --The ACK field of the segment is set to the parameter value.
        to_NET.seg.ack_flag := TRUE;
        to_NET.seg.ack_num  := parameter;

   --Fill in the rest of the segment and network parameters.
        to_NET.seg.seq_num      := sv.send_next;
        to_NET.seg.rst_flag     := FALSE;
        to_NET.seg.syn_flag     := FALSE;
        to_NET.seg.push_flag    := FALSE;
        to_NET.seg.fin_flag     := FALSE;
        to_NET.seg.data_offset  := OPTIONLESS_HEADER;
        to_NET.seg.window       := sv.recv_wndw;

   --Add security and precedence information to header.
   --Add in urgent information if needed.
        if (sv.send_urg > to_NET.seg.seq_num)
        then  --record urgent data position in header
              to_NET.seg.urg_flag := TRUE;
              to_NET.seg.urgptr   := sv.send_urg - to_NET.seg.seq_num;
        else  to_NET.seg.urg_flag := FALSE;

        format_net_params( 0 );
        compute_checksum;
        TRANSFER to_NET to the network protocol entity;

   --Adjust implementation dependent ACK parameters such as
   --ACK timer, or state_vector element for the last ACK'd octet.
end;
```

6.3.6.3.35 send_fin The send_fin action procedure records a close request and, if no data is waiting to be transmitted, formats and sends an empty segment with the FIN flag set. If data is waiting and the window permits, the FIN is sent along with the data.
The data effects of this procedure are:

- Data examined: sv.send_next

- Data modified: sv.send_finflag sv.send_push

```
--Record the CLOSE service request.  The CLOSE implies a PUSH.
     sv.send_finflag := TRUE;
     sv.send_push := sv.send_next;

--The FIN is sent along with any waiting data.
     send_new_data;
```

6.3.6.3.36 send new data The send_new_data action procedure examines the send window, the amount of pushed data, and segment size restrictions to determine if any waiting data can be sent to the remote TCP. The data effects of this procedure are:

- Date examined:
  - sv.send_max_seg
  - sv.source_port
  - sv.destination_port
  - sv.recv_next
  - sv.recv_wndw
  - sv.send_finflag
- Data modified:
  - sv.send_next
  - sv.send_free
  - sv.send_wndw
  - sv.send_push
  - sv.send_urg
  - all fields of to_NET
- Local variables: send_amount

```
begin
--The amount of data to be sent is determined by the
--send window, the amount of data waiting, the amount of
--pushed data, and segment size restrictions.

  if ((sv.send_wndw !!= 0) and (sv.send_next !!= sv.send_free))
  then begin
       --Data can be sent, but how much?
       --Check for pushed data, which must be sent as soon
       --as the window allows.
         begin
         if (sv.send_push > sv.send_next)
         then --Pushed data awaits transmission

             if (sv.send_push < sv.send_una + sv.send_wndw)
             then --all pushed data can be sent
                  send_amount := sv.send_push - sv.send_next;
                  to_ULP.seg.push_flag := TRUE;
             else --send all pushed data allowed by send window
                  send_amount := sv.send_una + sv.send_wndw - sv.send_next;
                  to_NET.seg.push_flag := FALSE;
         else --No pushed data waiting.  Refer to send policy
              --to determine amount (if any) to be sent.
              send_amount := send_policy();
              to_NET.seg.push_flag := FALSE;

       --How much data to send has been determined. Now
       --format and transmit the segment.
         if (send_amount > 0)
         then begin
            to_NET.seg.seq_num     := sv.send_next;
            to_NET.seg.ack_num     := sv.recv_next;
            to_NET.seg.ack_flag    := TRUE;
            to_NET.seg.syn_flag    := FALSE;
            to_NET.seg.rst_flag    := FALSE;
```

```
            to_NET.seg.data_offset := OPTIONLESS_HEADER;
            to_NET.seg.window      := sv.recv_wndw;
            --Add security and precedence to header.
            --The ULP may have already CLOSE-d.  If so, and this
            --data includes the last octet, set the FIN.
            if ((sv.send_finflag = TRUE) and
                 (sv.send_free = to_NET.seg.seq_num + send_amount))
            then to_NET.seg.fin_flag    := TRUE
            else to_NET.seg.fin_flag    := FALSE;

            if (sv.send_urg > to_NET.seg.seq_num)
            then --record urgent data position in header
                 to_NET.seg.urg_flag := TRUE;
                 to_NET.seg.urgptr := sv.send_urg - to_NET.seg.seq_num;
            else to_NET.seg.urg_flag := FALSE;

            dm_copy_from_send( sv.send_next, send_amount );
            sv.send_next := sv.send_next + send_amount;

            format_net_params( send_amount );
            compute_checksum;
            TRANSFER to_NET to the network protocol entity;

            --Depending on the retransmission policy chosen for
            --an implementation, a retransmission timer
            --may now need to be set for the newly sent data.
                 --implementation dependent action

            end; --of preparation and transmission of data.
     end;
end;
```

6.3.6.3.37 send policy Barring pushed data and zero receive windows, the TCP entity is left to segment and transfer data at its convenience. The number of octets that should be sent beginning at sv.send_next is returned to the calling procedure.

The definition of "convenience" should be influenced by design goals. If the primary goal is low overhead in terms of segment generation, then data should be accumulated until a maximum segment's worth (defined by the remote TCP) is ready. However, if quick response is the main goal, the TCP entity should segment and transmit data at regular intervals to minimize delay.

Another aspect of the send policy is related to window management. Discussed in the Section 6.1.2, the handling of small send windows may alter sending behavior. The TCP entity may choose to avoid sending into small windows (where small is defined as a percentage of segment size or storage capacity) to achieve better throughput.

6.3.6.3.38 set_fin The set_fin action procedure records the presence of a FIN in an incoming segment. The ULP is informed of the remote ULP's CLOSE after all data from the remote ULP is delivered.
The data effects of this procedure are:

- Data examined: sv.recv_save sv.recv_next

- Data modified: sv.recv_finflag

```
begin
     --Record the FIN's presence for use in the ESTABLISHED state.
          sv.recv_finflag := TRUE;
          sv.recv_push := sv.recv_next;

     --If no data is waiting to be delivered, a CLOSING
     --service response is issued to inform the local ULP of the
     --remote ULP's CLOSE request.

          if (sv.recv_save = sv.recv_next)
              and (no data is awaiting re-ordering)
          then
               to_ULP.service_response := CLOSING;
               to_ULP.lcn := sv.lcn;
               TRANSFER to_ULP to the ULP named by sv.source_port.
end;
```

6.3.6.3.39 start_time_wait The start_time_wait action procedure cancels all other timers and sets the final "TIME_WAIT" timer which allows time for the final FIN acknowledgement to reach the remote TCP before clearing the state vector of this connection.
The data effects of this procedure are:

- Data examined: - none -

- Data modified: - none -

```
     begin
       --Issue timer cancellation requests to the execution environment
       --corresponding to all current timers.
           cancel_timer( ULP_TIMEOUT );
           cancel_timer( RETRANSMIT );

           --Depending on implementation strategies, ACK timers and
           --zero window timers may also exist.

       --Start up the time_wait timer for the appropriate duration--currently
       --suggested to be 2 minutes.

           start_timer( TIME_WAIT, TIME_WAIT_INTERVAL );
     end;
```

6.3.6.3.40 update The update routine takes a new ACK from the incoming segment to update the send and receive variables. The data effects of this procedure are:

- Data examined:
  from_NET.seg.ack_num from_NET.seg.window
  from_NET.seg.seq_num

- Data modified:
  sv.send_una sv.send_wndw
  sv.send_lastup1 sv.send_lastup2

```
begin
--Take only new ACKs, i.e. those greater than sv.send_una.

  if from_NET.seg.ack_num > sv.send_una
  then begin   --update the retransmission queue
       dm_remove_from_send(sv.send_una,(from_NET.seg.ack_num - sv.send_una));
       sv.send_una := from_NET.seg.ack_num;

       --Depending on retransmission strategy, the retransmission
       --timer may need resetting because of the new ACK.
            --implementation dependent action

       --The retransmission timeout interval may need adjustment
       --to adapt to the round-trip time of the data just ACK-ed.
            --implementation dependent action

       --The ULP timeout timer may need resetting due to the
       --the successful delivery of the newly ACK-ed data.
            --implementation dependent action
       end;

--A new window is provided if either the sequence number of this
--segment is newer than the one last used to update the window, or
--(for 1-way data transfer) the sequence number is the same but
--the ACK is greater.

  if ((sv.send_lastup1 < from_NET.seg.seq_num) or
            (sv.send_lastup2 < from_NET.seg.ack_num))
  then begin
       sv.send_wndw := (from_NET.seg.ack_num + from_NET.seg.window)
                                          - sv.send_una;
       sv.send_lastup1 := from_NET.seg.seq_num;
       sv.send_lastup2 := from_NET.seg.ack_num;

       --Because a new send window has arrived, try to send data.
         send_new_data;

       end;
end;
```

7. EXECUTION ENVIRONMENT REQUIREMENTS

Throughout this document, the environmental model portrays each protocol entity acting as an independent process. Within this model, the execution environment must provide two facilities: inter-process communication and timing.

7.1 INTER-PROCESS COMMUNICATION

The execution environment must provide an inter-process communication facility to enable independent processes to pass units of information, called messages. For TCP's purposes, the IPC facility is required to preserve the order of messages.

TCP uses the IPC facility to exchange interface parameters and data with upper layer protocols across its upper interface and the network protocol across the lower interface. Sections 3 and 5 specify these interfaces.

In the service and entity specifications, this service is accessed through a the following primitive:

1. TRANSFER - passes a message to a named target process.

7.2 TIMING

The execution environment must provide a timing facility that maintains 32-bit clock (possibly fictitious) with units no coarser than 1 second. A process must be able to set a timer for a specific time period and be informed by the execution environment when the time period has elapsed. A process must also be able to cancel a previously set timer.

Several TCP mechanisms use the timing facility. The positive acknowledgement with retransmission mechanism uses timers to ensure that if data or acknowledgements are lost, they are re-sent. The ULP timeout mechanism uses the timing facility to clock the delay between data transmission and acknowledgement. The time-wait mechanism uses a timer to allow enough time for a final FIN acknowledgement to arrive at the remote TCP entity before connection termination. Other uses for a timing facility are implementation dependent.

In the upper service and entity specification, the timing services are accessed with the following primitives:

1. SET_TIMER(timer_name, time_interval) - allows a given interval of time and an identifier to be specified. After the specified interval elapse, and timeout indication and the identifier is returned to the issuing process.

2. CANCEL_TIMER(timer_name) - allows the timeout associated with the identifier to be terminated.

3. CURRENT_TIME - returns the current time.

8. GLOSSARY

Acknowledgement Number
A 32-bit field of the TCP header containing the next sequence number expected by the sender of the segment.

ACK
Acknowledgement flag: a control bit in the TCP header indicating that the acknowledgement number field is significant for this segment.

Checksum
A 16-bit field of the TCP header carrying the one's complement based checksum of both the header and data in the segment.

connection
A logical communication path identified by a pair of sockets.

datagram
A self-contained package of data carrying enough information to be routed from source to destination without reliance on earlier exchanges between source or destination and the transporting network.

datagram service
A datagram, defined above, delivered in such a way that the receiver can determine the boundaries of the datagram as it was entered by the source. A datagram is delivered with high probability to the desired destination, but it may possibly be lost. The sequence in which datagrams are entered into the network by a source is not necessarily preserved upon delivery at the destination.

Data Offset
A TCP header field containing the number of 32-bit words in the TCP header.

Destination Address
The destination address, usually the network and host identifiers. Although not carried in the TCP header, this value is passed to and received from the network protocol entity with each segment.

Destination Port
The TCP header field containing a 2-octet value identifying the destination upper level protocol of a segment's data.

FIN
A control bit of the TCP header indicating that no more data will be sent by the sender.

header
The collection of control information transmitted with data between peer entities.

host
A computer, particularly a source or destination of messages from the point of

view of the communication network.

Identification
A value passed with each segment to the network protocol entity (Internet Protocol). This identifying value assigned by the sending TCP aids in assembling the fragments of a datagram.

internetwork
A set of interconnected subnetworks.

internet address
A four octet (32 bit) source or destination address composed of a Network field and a Local Address field.

internet datagram
The package exchanged between a pair of IP modules. It is made up of an internet header and a data portion.

IP
Internet Protocol

ISN
The Initial Sequence Number. The first sequence number used for either sending or receiving on a connection. It is selected on a clock based procedure.

local network
The network directly attached to host or gateway.

module
An implementation, usually in software, of a protocol or other procedure.

MSL
Maximum Segment Lifetime, the time a TCP segment can exist in the internetwork system. Arbitrarily defined to be 2 minutes.

octet
An eight bit byte.

Options
The optional set of fields at the end of the TCP header used in a SYN segment to carry the maximum segment size acceptable to the sender.

packet
The unit of data transmitted by a packet-switched network. A packet usually contains nested control information and data from the higher layer protocols using the network.

packet network
A network based on packet-switching technology. Messages are split into small units (packets) to be routed independently on a store and forward basis. This packetizing pipelines packet transmission to effectively use circuit bandwidth.

Padding
A header field inserted after option fields to ensure that the data portion begins on a 32-bit word boundary. The padding field value is zero.

port
The portion of a socket that specifies the upper level protocol associated with the data.

precedence
A measure of datagram importance. It is one of the service quality parameters supported by the Internet Protocol's type of service mechanism.

PUSH
A control bit of the TCP header occupying no sequence space, indicating that this segment contains data that must be pushed through to the receiving ULP.

push service
A service provided by TCP to the upper level protocols. A push directs TCP to segment, send, and deliver data received up to that point as soon as flow control permits.

receive next sequence number
The next sequence number a TCP is expecting to receive.

receive window
This represents the sequence numbers a TCP is willing to receive. Thus, the TCP considers that segments overlapping the range RECV_NEXT to RECV_NEXT + RECV_WND - 1 carry acceptable data or control. Segments containing sequence numbers entirely outside of this range are considered duplicates and discarded.

reliability
A quality of data transmission defined as guaranteed, ordered delivery.

Reserved
A 6-bit field of the TCP header that is not currently used but must be zero.

RST
A control bit of the TCP header indicating that the connection associated with this segment is to be terminated.

segment
The unit of data exchanged by TCP modules. This term may also be used to describe the unit of exchange between any transport protocol modules.

segment length
The amount of sequence number space occupied by a segment, including any controls which occupy sequence space.

send sequence
This is the next sequence number the TCP will use to send data on the connection. It is initially selected from an initial sequence number curve (ISN)

and is incremented for each octet of data or sequenced control transmitted.

send window
This represents the sequence numbers which the remote TCP is willing to receive. It is the value of the window field specified in segments from the remote (data receiving) TCP. The range of new sequence numbers which may be emitted by a TCP lies between SEND_NEXT and SEND_UNA + SEND_WNDW - 1. (Retransmissions of sequence numbers between SEND_UNA and SND_NEXT are expected, of course.)

Sequence Number
A 32-bit field of the TCP header containing the sequence number of the 1) a sequenced control flag (if present), or 2) the first byte of data (if present), or, 3) for empty segments, the sequence number of the next data octet to be sent.

socket
An address which specifically includes a port identifier, that is, the concatenation of an Internet Address with a TCP port.

Source Port
The TCP header field containing a 2-octet value identifying the source upper level protocol of a segment's data.

TCP segment
The package exchanged between TCP modules made up of the TCP header and a text portion (which may be empty).

ULP
Upper Level Protocol: any protocol above TCP in the layered protocol hierarchy that uses TCP. This term includes presentation layer protocols, session layer protocols, and user applications.

Urgent Pointer
A TCP header field containing a positive offset to the sequence number of the segment indicating the position of urgent data in the connection's data stream. This field is valid only when the URG flag is on.

URG
A control bit of the TCP header indicating that the urgent field contains a valid pointer to urgent information in the connection's data stream.

Window
A 2-octet field of the TCP header indicating the number of data octets (relative to the acknowledgement number in the header) that the segment sender is currently willing to accept.

9. BIBLIOGRAPHY

1. V. Cerf and R. Kahn, "A Protocol for Packet Network Interconnection," IEEE Transactions on Communications, May 1974.

2. J. Fletcher and R. Watson, "Mechanisms for a Reliable Timer-Based Protocol," Computer Networks vol.2, 1978, pp.271-290.

3. J. Postel (ed.), "DoD Standard Internet Protocol," Defense Advanced Research Projects Agency, Information Processing Techniques Office, RFC791, September, 1981.

4. J. Postel (ed.), "DoD Standard Transmission Control Protocol," Defense Advanced Research Projects Agency, Information Processing Techniques Office, RFC793, September, 1981.

5. J. Postel, "Assigned Numbers," RFC 790, USC/Information Sciences Institute, September, 1981.

6. SDC,"DoD Protocol Reference Model," SDC TM-7172/201/00, Contract No. DCA100-82-C-0036, February 1982.

7. SDC, "Protocol Specification Report," SDC TM-7172/301/00, Contract No. DCA100-82-C-0036, March 1982.

8. SDC, "Proposed Internetwork Protocol Standard,"SDC TM-7172/481/00, Contract No. DCA100-82-C-0036, July 1982.

9. C. Sunshine, "Efficiency of Interprocess Communication Protocols for Computer Networks," IEEE Transactions on Communications, February 1977, pp. 287-293.

10. C. Sunshine and Y.Dalal, "Connection Management in Transport Protocols," Computer Networks vol.2, 1978, pp.454-473.

APPENDIX A: Retransmission Strategy Effectiveness

As noted in the entity overview, Section 6.1, a TCP implementation may employ one of several retransmission strategies:

a. First-only retransmission - A TCP maintains one retransmission timer for the queue, retransmitting the front segment (or segment's worth of data) when the timer expires.

b. Batch retransmission - A TCP maintains one retransmission timer for the queue, retransmitting all segments on the queue when the timer expires.

c. Individual retransmission - A TCP maintains one timer per segment on the queue, retransmitting each segment when its individual timer expires.

The first-only retransmission strategy is efficient in terms traffic generated because only lost segments are retransmitted; but the strategy can cause long delays. The batch retransmission creates more traffic but decreases the likelihood of long delays. The individual retransmission strategy is a compromise between delay and traffic but requires much more processing time from the TCP entity. However, the actual effectiveness of each scheme depends in part on the acceptance policy (Section 6.1.3) of the receiving TCP.

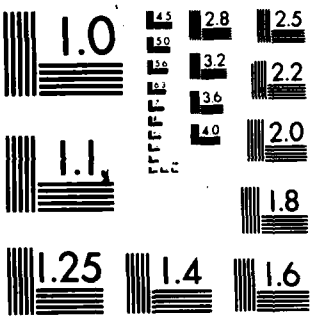
For example, suppose a sending TCP sends three segments, all within the send window, to a receiving TCP. The first segment is lost by the network. A receiving TCP using the "in-order" acceptance strategy discards the second and third segments. A receiving TCP using the "in-window" strategy accepts the second and third segments, but does not acknowledge or deliver any data until the intervening segment arrives.

Batch retransmission performs better with the in-order acceptance strategy because the receiving TCP has discarded all segments. All three segments must be retransmitted--the sooner the better. First-only retransmission performs better with the in-window acceptance policy because only the necessary retransmissions occur since the receiving TCP has kept the segments within its receive window and awaits only the lost segment.

Unfortunately, a sending TCP cannot know what acceptance policy is being used by the receiving TCP. Instead, the retransmission strategy must be chosen according implementation dependent and configuration dependent design goals.

AD-A126 559 PROPOSED DOD (DEPARTMENT OF DEFENSE) TRANSMISSION CONTROL PROTOCOL STANDARD(U) SYSTEM DEVELOPMENT CORP SANTA MONICA CA 06 JUL 82 SDC-TM-7172/482/00

UNCLASSIFIED DCA100-82-C-0036 F/G 17/2 NL



APPENDIX B: Dynamic Retransmission Timer Computation

Because of the variability of the networks that compose and internetwork system and the wide range of uses of TCP connections, the retransmission timeout should be dynamicaliy determined. One procedure for determining a retransmission time out is give her as an illustration.

Measure the elapsed time between sending a data octet with a particular sequence number and receiving an acknowledgement that covers that sequence number. (Segments sent do not have to match segments received.) This measured elapsed time is the Round Trip Time. Next, compute a Smoothed Round Trip Time (SRTT) as:

SRTT = (ALPHA * SRTT) + ((1-ALPHA) * RTT)

and based on this, compute the retransmission timeout (RTO) as:

RTO = minimum(UBOUND, maximum(LBOUND,(BETA*SRTT))))

where:

UNBOUND = an upper bound on the timeout (e.g., 1 minute)
LBOUND = a lower bound on the timeout (e.g., 1 second)
ALPHA = a smoothing factor (e.g., .8 to .9)
BETA = a delay variance factor (e.g., 1.3 to 2.0)

APPENDIX C: Alternatives in Service Interface Primitives

The service primitives offered to the upper level protocol are specified in section 3.1. The service request primitives are:

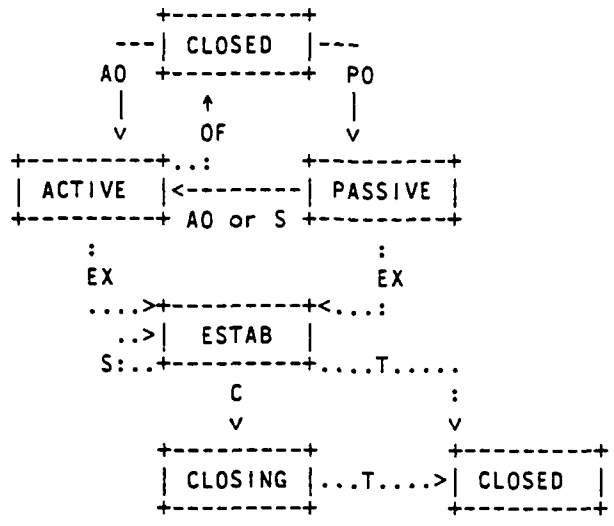
- Unspecified Passive Open,
- Fully Specified Passive Open,
- Active Open
- Active Open with Data,
- Send,
- Allocate,
- Status,
- Close, and
- Abort

These primitives support the minimal services required of a TCP. However, combinations or modifications may offer additional services that are tailored to the requirements of a particular set of upper level protocols. Several examples are provided below.

If the protocol supporting TCP is the Internet Protocol and a TCP implementation wishes to export IP's option services [3] (including source routing, record routing, stream identification and timestamps), an addtional "options" parameter would be required in all Open and Send service requests.

An upper level protocol may need a reliable transaction service. That is, a ULP may wish to open a connection, send a single message, and then close the connection. To access this service, the specified service interface requires the ULP to issue at least two service primitives, an Open with Data and a Close, to excercise this service. A TCP may be designed with a service primitive that combined the Open and Close to form a new primitive, called perhaps Transaction, which would include all the Open parameters, the data to be transmitted, and the signal to close the connection after data delivery.

The upper layer service definition (Section 3.2) does not allow a Passive Open request to be followed by an Active Open request. Instead, the ULP must first issue a Close or Abort request to cancel the Passive Open request, then issue an Active Open request. A TCP may be designed to allow "conversion" of open requests from passive to active. In this case, a ULP could issue a Full Passive Open request followed by an Active Open or a Send request to actively initiate a connection. Thus, the local entity service diagram (appearing in Section 3.2) changes to include a transition from the PASSIVE to the ACTIVE state as shown below:



TCP LOCAL SERVICE STATE MACHINE SUMMARY